Gerard O'Regan

Mathematics in Computing

An Accessible Guide
to Historical, Foundational
and Application Contexts

Springer

Mathematics in Computing

Gerard O'Regan

Mathematics in Computing

An Accessible Guide to Historical,
Foundational and Application Contexts

Springer

Gerard O'Regan
Mallow, Ireland

To my siblings
Mary Rose, Donal, Francis and Marguerita

Preface

Overview

The objective of this book is to give the reader a flavour of mathematics used in the computing field. The goal is to show how mathematics is applied in computing, rather than the study of mathematics for its own sake.

Organization and Features

The first chapter discusses the contributions made by early civilisations to computing. This includes work done by the Babylonians, Egyptians and Greeks. The Egyptians applied mathematics to solve practical problems such as the construction of pyramids. The Greeks made a major contribution to mathematics and geometry, and most students are familiar with the work of Euclid.

Chapter 2 provides an introduction to fundamental building blocks in mathematics including sets, relations and functions. A set is a collection of well-defined objects and it may be finite or infinite. A relation between two sets A and B indicates a releationship between members of the two sets, and is a subset of the Cartesian product of the two sets. A function is a special type of relation such that for each element in A there is at the most one element in the co-domain B. Functions may be partial or total and injective, surjective or bijective.

Chapter 3 provides an introduction to logic including propositional and predicate logic. The nature of mathematical proof is discussed.

Chapter 4 provides an introduction to the important field of software engineering. The birth of the discipline was at the Garmisch conference in Germany in the late 1960s. The extent to which mathematics should be employed in software engineering is discussed, and this remains a topic of active debate.

Chapter 5 discusses formal methods, which consist of a set of mathematical techniques to specify and derive a program from its specification. Formal methods may be employed to rigorously state the requirements of the proposed system; they may be employed to derive a program from its mathematical specification; and they

provide a rigorous proof that the implemented program satisfies its specification. They have been mainly applied to the safety critical field.

Chapter 6 presents the Z specification language, which is one of the most widely used formal methods. It was developed at Oxford University in the UK.

Chapter 7 presents the fundamentals of number theory, and discusses prime number theory and the greatest common divisor and least common multiple of two numbers.

Chapter 8 discusses cryptography, which is an important application of number theory. The codebreaking work done at Bletchley Park in England during the Second World War is discussed, and the fundamentals of cryptography, including private and public key cryptosystems, are discussed.

Chapter 9 presents coding theory and is concerned with error detection and error correction codes. The underlying mathematics is discussed, and this includes abstract mathematics such as group theory, rings, fields, and vector spaces.

Chapter 10 discusses language theory and includes a discussion on grammar, parse trees, and derivations from a grammar. The important area of programming language semantics is discussed, including an overview of axiomatic, denotational and operational semantics.

Chapter 11 discusses computability and decideability. The Church-Turing thesis states that anything that is computable is computable by a Turing machine. Church and Turing showed that mathematics is not decideable. In other words, there is no mechanical procedure (i.e., algorithm) to determine whether an arbitrary mathematical proposition is true or false, and so the only way is to determine the truth or falsity of a statement is try to solve the problem.

Chapter 12 discusses probability and statistics and includes a discussion on discrete and continuous random variables, probability distributions, sample spaces, sampling, the abuse of statistics, variance and standard deviation, and hypothesis testing. The application of probability to the software reliability field is discussed.

Chapter 13 discusses matrices including 2×2 and general $n \times m$ matrices. Various operations such as the addition and multiplication of matrices are considered, and the determinant and inverse of a matrix is discussed. The application of matrices to solve a set of linear equations using Gaussian elimination is cosidered.

Chapter 14 discusses complex numbers and quaternions. Complex numbers of the form $a + bi$ where a and b are real numbers, and $i^2 = -1$. Quaternions are a generalization of complex numbers to quadruples that satisfy the quaternion formula $i^2 = j^2 = k^2 = -1$.

Chapter 15 provides a very short introduction to calculus, and provides a high-level overview of limits, continuity, differentiation, integration, and numerical analysis. Fourier series, Laplace transforms and differential equations are briefly discussed.

Chapter 16 discusses graph theory where a graph $G = (V,E)$ consists of vertices and edges. It is a practical branch of mathematics that deals with the arrangements of vertices and the edges between them. It has been applied to practical problems such as the modeling of computer networks, determining the shortest driving route between two cities, and the traveling salesman problem.

Audience

The audience of this book includes computer science students who wish to obtain an overview of mathematics used in computing, and mathematicians who wish to get an overview of how mathematics is applied in the computing field. The book will also be of interest to the motivated general reader.

Acknowledgments

I am deeply indebted to my family and friends who supported my efforts in this endeavour.

Cork, Ireland   Gerard O'Regan

Contents

List of Figures

Chapter 1
Mathematics in Civilization

> **Key Topics**
> Babylonian Mathematics
> Egyptian Civilisation
> Greek and Roman Civilisation
> Counting and Numbers
> Solving Practical Problems
> Syllogistic Logic
> Algorithms
> Early Ciphers

1.1 Introduction

It is difficult to think of western society today without modern technology. The last decades of the twentieth century have witnessed a proliferation of high-tech computers, mobile phones, text messaging, the Internet and the World Wide Web. Software is now pervasive, and it is an integral part of automobiles, airplanes, televisions and mobile communication. The pace of change as a result of all this new technology has been extraordinary. Today, consumers may book flights over the World Wide Web as well as keep in contact with family members in any part of the world via e-mail, Facebook, Skype or mobile phone. In previous generations, communication often involved writing letters that took months to reach the recipient.

Communication improved with the telegrams and the telephone in the late nineteenth century. Communication today is instantaneous with text messaging, mobile phones and e-mail, and the new generation probably views the world of their parents and grandparents as being old-fashioned.

The new technologies have led to major benefits[1] to society and to improvements in the standard of living for many citizens in the western world. It has also reduced

---

[1] Of course, it is essential that the population of the world moves towards more sustainable development to ensure the long-term survival of the planet for future generations. This involves finding technological and other solutions to reduce greenhouse gas emissions as well as moving to

the necessity for humans to perform some of the more tedious or dangerous manual tasks, as computers may now automate many of these. The increase in productivity due to the more advanced computerised technologies has allowed humans, at least in theory, the freedom to engage in more creative and rewarding tasks.

Early societies had a limited vocabulary for counting, e.g. 'one, two, three, many' which is associated with some primitive societies, and indicates limited computation and scientific ability. It suggests that there was no need for more sophisticated arithmetic in the primitive culture as the problems dealt with were elementary. These early societies would typically have employed their fingers for counting, and as humans have five fingers on each hand and five toes on each foot then the obvious bases would have been 5, 10 and 20. Traces of the earlier use of the base 20 system are still apparent in modern languages such as English and French. This includes phrases such as 'three score' in English and '*quatre vingt*' in French.

The decimal system (base 10) is used today in western society, but the base 60 was common in computation *circa* 1500 B.C. One example of the use of base 60 today is the sub-division of hours into 60 minutes, and the sub-division of minutes into 60 seconds. The base 60 system (i.e. the sexagesimal system) is inherited from the Babylonians [Res:84]. The Babylonians were able to represent arbitrarily large numbers or fractions with just two symbols. The binary (base 2) and hexadecimal (base 16) systems play a key role in computing (as the machine instructions that computers understand are in binary code).

The achievements of some of these ancient societies were spectacular. The archaeological remains of ancient Egypt such as the pyramids at Giza and the temples of Karnak and Abu Simbal are impressive. These monuments provide an indication of the engineering sophistication of the ancient Egyptian civilisation. The objects found in the tomb of Tutankamun[2] are now displayed in the Egyptian museum in Cairo, and demonstrate the artistic skill of the Egyptians.

The Greeks made major contributions to western civilisation including contributions to mathematics, philosophy, logic, drama, architecture, biology and democracy.[3] The Greek philosophers considered fundamental questions such as ethics, the nature of being, how to live a good life, and the nature of justice and

---

a carbon-neutral way of life. The solution to the environmental issues will be a major challenge for the twenty-first century.

[2] Tutankamun was a minor Egyptian pharaoh who reigned after the controversial rule of Akenaten. Tutankamun's tomb was discovered by Howard Carter in the Valley of the Kings, and the tomb was intact. The quality of the workmanship of the artefacts found in the tomb is extraordinary and a visit to the Egyptian museum in Cairo is memorable.

[3] The origin of the word "democracy" is from demos ($\delta\eta\mu o\varsigma$) meaning people and kratos ($\kappa\rho\alpha\tau o\varsigma$) meaning rule. That is, it means rule by the people. It was introduced into Athens following the reforms introduced by Cleisthenes. He divided the Athenian city state into thirty areas. Twenty of these areas were inland or along the coast and ten were in Attica itself. Fishermen lived mainly in the ten coastal areas, farmers in the ten inland areas, and various tradesmen in Attica. Cleisthenes introduced ten new clans where the members of each clan came from one coastal area, one inland area on one area in Attica. He then introduced a Boule (or assembly) which consisted of 500 members (50 from each clan). Each clan ruled for 1/10th of the year.

politics. The Greek philosophers include Parmenides, Heraclitus, Socrates, Plato and Aristotle. The Greeks invented democracy which, however, was radically different from today's representative democracy.[4] The sophistication of Greek architecture and sculpture is evident from the Parthenon on the Acropolis, and the Elgin marbles[5] that are housed today in the British Museum, London.

The Hellenistic[6] period commenced with Alexander the Great and led to the spread of Greek culture throughout most of the known world. The city of Alexandria became a centre of learning and knowledge during the Hellenistic period. Its scholars included Euclid who provided a systematic foundation for geometry. His work is known as "The Elements", and it consists of 13 books. The early books are concerned with the construction of geometric figures, number theory and solid geometry.

There are many words of Greek origin that are part of the English language. These include words such as "psychology" which is derived from two Greek words, *psyche* ($\psi\upsilon\chi\varepsilon$) and *logos* ($\lambda o\gamma o\varsigma$). The Greek word '*psyche*' means mind or soul, and the word '*logos*' means an account or discourse. Other examples are anthropology derived from '*anthropos* ($\alpha\nu\tau\rho o\pi o\varsigma$) and '*logos*' ($\lambda o\gamma o\varsigma$).

The Romans were influenced by Greeks culture. The Romans built aqueducts, viaducts, and amphitheatres. They also developed the Julian calendar, formulated laws (*lex*), and maintained peace throughout the Roman Empire (*pax Romano*). The ruins of Pompeii and Herculaneum demonstrate their engineering capability. Their numbering system is still employed in clocks and for page numbering in documents. However, it is cumbersome for serious computation. The collapse of the Roman Empire in Western Europe led to a decline in knowledge and learning in Europe. However, the eastern part of the Roman Empire continued at Constantinople until its sacking by the Ottomans in 1453.

1.2 The Babylonians

The Babylonian[7] civilisation flourished in Mesopotamia (in modern Iraq) from about 2000 B.C. until about 300 B.C. Various clay cuneiform tablets containing mathematical texts were discovered and later deciphered in the nineteenth century [Smi:23]. These

---

[4] The Athenian democracy involved the full participations of the citizens (i.e. the male adult members of the city state who were not slaves) whereas in representative democracy the citizens elect representatives to rule and represent their interests. The Athenian democracy was chaotic and could also be easily influenced by individuals who were skilled in rhetoric. There were teachers (known as the Sophists) who taught wealthy citizens rhetoric in return for a fee. The origin of the word "sophist" is the Greek word $\sigma o\phi o\varsigma$ meaning wisdom. One of the most well known of the sophists was Protagorus. The problems with the Athenian democracy led philosophers such as Plato to consider alternate solutions such as rule by philosopher kings. This totalitarian utopian state is described in Plato's Republic.

[5] The Elgin marbles are named after Lord Elgin who moved them from the Parthenon in Athens to London in 1806. The marbles show the Pan-Athenaic festival that was held in Athens in honour of the goddess Athena after whom Athens is named.

[6] The origin of the word Hellenistic is from Hellene ($E\lambda\lambda\eta\nu$) meaning Greek.

[7] The hanging gardens of Babylon were one of the seven wonders of the ancient world.

included tables for multiplication, division, squares, cubes and square roots, and the measurement of area and length. Their calculations allowed the solution of a linear equation and one root of a quadratic equation to be determined. The late Babylonian period (*circa* 300 B.C.) includes work on astronomy.

They recorded their mathematics on soft clay using a wedge shaped instrument to form impressions of the *cuneiform* numbers. The clay tablets were then baked in an oven or by the heat of the sun. They employed just two symbols (1 and 10) to represent numbers, and these symbols were then combined to form all other numbers. They employed a positional number system[8] and used the base 60 system. The symbol representing 1 could also (depending on the context) represent 60, 60^2, 60^3, etc. It could also mean 1/60, 1/3,600, and so on. There was no zero employed in the system and there was no decimal point (no "sexagesimal point"), and therefore the context was essential.

$$ \text{Ⲏ ⊲ Ⲏ} $$

The example above illustrates the cuneiform notation and represents the number $60 + 10 + 1 = 71$. The Babylonians used the base 60 system for computation, and this base is still in use today in the division of hours into minutes and the division of minutes into seconds. One possible explanation for the use of the base 60 notation is the ease of dividing 60 into parts. It is divisible by 2, 3, 4, 5, 6, 10, 12, 15, 20 and 30. They were able to represent large and small numbers and had no difficulty in working with fractions (in base 60) and in multiplying fractions. The Babylonians maintained tables of reciprocals (i.e. $1/n$, $n = 1 \ldots, 59$) apart from numbers like 7, 11, etc., which cannot be written as a finite sexagesimal expansion (i.e. 7, 11, etc., are not of the form $2^\alpha 3^\beta 5^\gamma$).

The modern sexagesimal notation [Res:84] 1; 24, 51, 10 represents the number

$$1 + 24/60 + 51/3{,}600 + 10/216{,}000 = 1 + 0.4 + 0.0141666 + 0.0000462$$
$$= 1.4142129.$$

This is the Babylonian representation of the square root of 2. They performed multiplication as follows, e.g. consider $20 \times \text{sqrt}(2) = (20) \times (1; 24,51,10)$:

$$20 \times 1 = 20$$

$$20 \times ; 24 = 20 \times \frac{24}{60} = 8$$

$$20 \times \frac{51}{3{,}600} = \frac{51}{180} = \frac{17}{60} = ; 17$$

$$20 \times \frac{10}{216{,}000} = \frac{3}{3{,}600} + \frac{20}{216{,}000} = ; 0,3,20$$

[8] A positional numbering system is a number system where each position is related to the next by a constant multiplier. The decimal system is an example $546 = 5 \times 10^2 + 4 \times 10^1 + 6$.
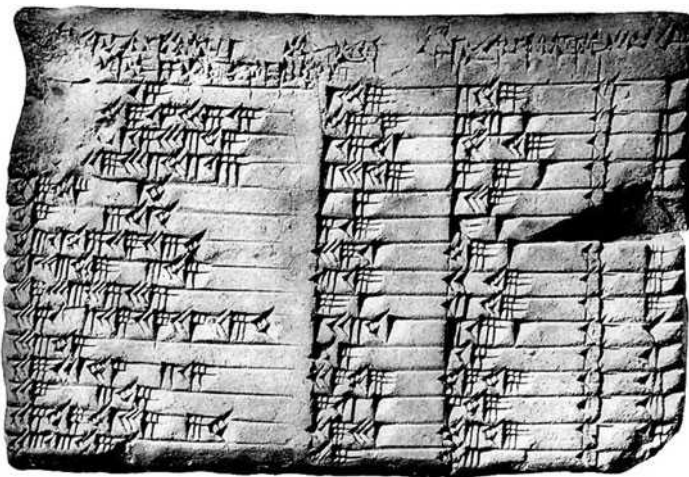
Fig. 1.1 The Plimpton 322 tablet

Fig. 1.2 Geometric representation of $(a + b)^2 = (a^2 + 2ab + b^2)$
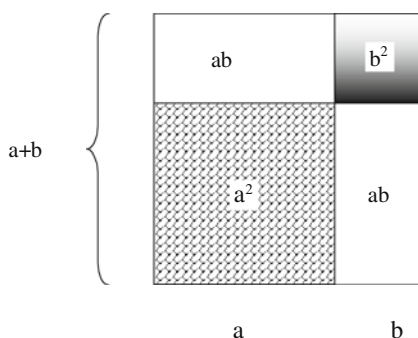
Hence, the product $20 \times sqrt(2) = 20; +8; +; 17+; 0,3,20 = 28; 17,3,20$.

The Babylonians appear to have been aware of Pythagoras's Theorem about 1,000 years before the time of Pythagoras. The Plimpton 322 tablet records various Pythagorean triples, i.e. triples of numbers (a, b, c) where $a^2 + b^2 = c^2$. It dates from approximately 1700 B.C. (Fig. 1.1).

They developed algebra to assist with problem solving, and their algebra allowed problems involving length, breadth and area to be discussed and solved. They did not employ notation for the representation of unknown values (e.g. let x be the length and y be the breadth), and instead they used words like 'length' and 'breadth'. They were familiar with square roots (and used them in their calculations) and techniques that allowed one root of a quadratic equation to be solved.

They were also familiar with various mathematical identities such as $(a + b)^2 = (a^2 + 2ab + b^2)$ as illustrated geometrically in Fig. 1.2. They worked on astronomical problems and had mathematical theories of the cosmos to make predictions of when

Fig. 1.3 Egyptian
representation of the number
276

eclipses and other astronomical events would occur. They were interested in astrology, and associated various deities with the heavenly bodies such as the planets, the sun and the moon. They associated various cluster of stars with familiar creatures such as lions, goats and so on.

The Babylonians used counting boards to assist with counting and simple calculations. A counting board is an early version of the abacus, and was usually made of wood or stone. It contained grooves that allowed beads or stones to move along the groove. The abacus differs from counting boards in that the beads in abaci contain holes that enable them to be placed in a particular rod of the abacus.

1.3   The Egyptians

The Egyptian Civilisation developed along the Nile from about 4000 B.C. and the pyramids were built around 2500 B.C. They used mathematics to solve practical problems such as measuring time, measuring the annual Nile flooding, calculating the area of land, book keeping and accounting and calculating taxes. They developed a calendar *circa* 3000 B.C., which consisted of 12 months with each month having 30 days. There were then five extra feast days to give 365 days in a year. Egyptian writing commenced around 3500 B.C. and is recorded on the walls of temples and tombs.[9] A reed-like parchment termed "papyrus" was used for writing, and three Egyptian writing scripts were employed. These were hieroglyphics, the hieratic script, and the demotic script.

For example, the representation of the number 276 in Egyptian hieroglyphics is given by (Fig. 1.3).

Hieroglyphs are little pictures and are used to represent words, alphabetic characters as well as syllables or sounds. Champollion did the deciphering of hieroglyphics with his work on the Rosetta stone that was discovered during the Napoleonic campaign in Egypt. The Rosetta stone is now in the British Museum in London. It contains three scripts, hieroglyphics, demotic script and Greek. The key to its decipherment was that the Rosetta stone contained just one name "Ptolemy" in the Greek text, and this was identified with the hieroglyphic characters in the cartouche[10] of the hieroglyphics. There was just one cartouche on the Rosetta stone, and Champollion inferred that the cartouche represented the name "Ptolemy". He was familiar with

---

[9] The decorations of the tombs in the Valley of the Kings record the life of the pharaoh including his exploits and successes in battle.

[10] The cartouche surrounded a group of hieroglyphic symbols enclosed by an oval shape. Champollion's insight was that the group of hieroglyphic symbols represented the name of the Ptolemaic pharaoh "Ptolemy".

Fig. 1.4 Egyptian numerals

Fig. 1.5 Egyptian
representation of the fraction
1/276
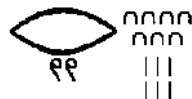
another multi-lingual object that contained two names in the cartouche. One name he recognised as Ptolemy and the other he deduced from the Greek text as "Cleopatra". This led to the breakthrough in the translation of the hieroglyphics [Res:84].

The Rhind Papyrus is a famous Egyptian papyrus on mathematics. The Scottish Egyptologist, Henry Rhind, purchased it in 1858. It is a copy created by an Egyptian scribe called Ahmose.[11] It is believed to date to 1832 B.C. and it contains examples of many kinds of arithmetic and geometric problems. Students may have used it as a textbook to develop their mathematical knowledge. This would have allowed them to participate in the pharaoh's building program.

The Egyptians were familiar with geometry, arithmetic and elementary algebra. They had techniques to find solutions to problems with one or two unknowns. A base 10 number system was employed with separate symbols for the numerals one, ten, a hundred, a thousand, a ten thousand, a hundred thousand, and so on. These hieroglyphic symbols are represented in Fig. 1.4.

The addition of two numerals is straightforward and involves adding the individual symbols, and where there are ten copies of a symbol it is then replaced by a single symbol of the next higher value. The Egyptian employed unit fractions (e.g. $1/n$ where n is an integer). These were represented in hieroglyphs by placing the symbol representing a "mouth" above the number. The symbol "mouth" represents part of. For example, the representation of the number 1/276 is shown in (Fig. 1.5).

The mathematical problems in the papyrus included the determination of the angle of the slope of the pyramid's face. The Egyptians were familiar with trigonometry including the fractions sine, cosine, tangent and cotangent, and knew how to build right angles into their structures by using the ratio 3:4:5. The papyrus also dealt with problems such as the calculation of the number of bricks required for part of a building project. Multiplication and division was cumbersome in Egyptian mathematics as they could only multiply and divide by two.

Suppose they wished to multiply a number n by 7. Then $n \times 7$ is determined by $n \times 2 + n \times 2 + n \times 2 + n$. Similarly, if they wished to divide 27 by 7 they would note that $7 \times 2 + 7 = 21$ and that $27 - 21 = 6$ and that therefore the answer was $3 \, {}^6/_7$.

---

[11] The Rhind papyrus is sometimes referred to as the Ahmes papyrus in honour of the scribe who wrote it in 1832 B.C.

Egyptian mathematics was cumbersome and the writing of their mathematics was long and repetitive. For example, they wrote a number such as 22 by $10 + 10 + 1 + 1$.

The Egyptians calculated the approximate area of a circle by calculating the area of a square 8/9 of the diameter of a circle. That is, instead of calculating the area in terms of our familiar πr^2 their approximate calculation yielded $(8/9 \times 2r)^2 = 256/81 r^2$ or $3.16\, r^2$. Their approximation of π was 256/81 or 3.16. They were able to calculate the area of a triangle and volumes. The Moscow papyrus includes a problem to calculate the volume of the frustum. The formula for the volume of a frustum of a square pyramid[12] was given by $V = 1/3 h(b_1^2 + b_1 b_2 + b_2^2)$ and when b_2 is 0 then the well-known formula for the volume of a pyramid is given, i.e. $1/3 h b_1^2$.

1.4   The Greeks

The Greeks made major contributions to western civilisation including mathematics, logic, astronomy, philosophy, politics, drama, and architecture. The Greek world of 500 B.C. consisted of several independent city-states such as Athens and Sparta, and various city-states in Asia Minor. The Greek polis (πολισ) or city-state tended to be quite small, and consisted of the Greek city and a certain amount of territory outside the city-state. Each city-state had political structures for its citizens, and these varied from one city-state to another. Some were oligarchs where political power was in the hands of a few individuals or aristocratic families. Others were ruled by tyrants (or sole rulers) who sometimes took power by force, but often had support from the public. The tyrants included people such as Solon, Peisistratus and Cleisthenes in Athens.

The reforms by Cleisthenes led to the introduction of the Athenian democracy. Power was placed in the hands of the male citizens (women or slaves did not participate in the Athenian democracy). It was an extremely liberal democracy where citizens voted on all-important issues. Often, this led to disastrous results as speakers who were skilled in rhetoric could exert significant influence. This led to Plato to advocate rule by philosopher kings and to reject democracy.

Early Greek mathematics commenced approximately 500–600 B.C. with work done by Pythagoras and Thales. Pythagoras was a philosopher and mathematician who had spent time in Egypt becoming familiar with Egyptian mathematics. He lived on the island of Samos and formed a secret society known as the Pythagoreans. They included men and women and believed in the transmigration of souls and that the number was the essence of all things. They discovered the mathematics harmony in music using the relationship between musical notes expressed in numerical ratios of small whole numbers. Pythagoras is credited with the discovery of Pythagoras's Theorem, although the Babylonians probably knew about this some 1,000 years

---

[12] The lengths of a side of the bottom base and that of the top base is b_1 and b_2

earlier. The Pythagorean society was dealt a major blow[13] by the discovery of the incommensurability of the square root of 2, i.e. there are no numbers p, q such that $\sqrt{2} = p/q$.

Thales was a sixth century B.C. philosopher from Miletus in Asia Minor who made contributions to philosophy, geometry and astronomy and his contributions to philosophy were mainly in the area of metaphysics, he was concerned with questions on the nature of the world. His objective was to give a natural or scientific explanation of the cosmos rather than rely on the traditional supernatural explanation of creation in Greek mythology. He believed that there was single substance that was the underlying constituent of the world, and he believed that this substance was water.

He also contributed to mathematics [AnL:95], and a well-known theorem in Euclidean geometry is named after him. This theorem states that if A, B and C are points on a circle such that AC is a diameter of the circle, then the angle $< ABC$ is a right angle.

The rise of Macedonia led to the Greek city-states being conquered by Philip of Macedonia in the fourth century B.C. His son, Alexander the Great, defeated the Persian Empire, and extended his empire to include most of the known world. This led to the Hellenistic Age with Greek language and culture spread throughout the known world. Alexander founded the city of Alexandra, and it became a major centre of learning. However, Alexander's reign was very short as he died at the young age of 33 in 323 B.C.

Euclid lived in Alexandria during the early Hellenistic period. He is considered the father of geometry and the *deductive method* in mathematics. His systematic treatment of geometry and number theory is published in the 13 books of the Elements [Hea:56]. It starts from 5 axioms, 5 postulates and 23 definitions to logically derive a comprehensive set of theorems. His method of proof was often *constructive* in that as well as demonstrating the truth of a theorem the proof would often include the construction of the required entity. He also used *indirect proof*, for example, that there are an infinite number of primes:

1. Suppose there is a finite number of primes (say n primes).
2. Multiply all n primes together and add 1 to form N.

$$(N = p_1 \times p_2 \times \ldots p_n + 1)$$

1. N is not divisible by p_1, p_2, ... , p_n as dividing by any of these gives a remainder of one.
2. Therefore, N must either be prime or divisible by some other prime that was not included in the list.
3. Therefore, there must be at least $n + 1$ primes.
4. This is a contradiction as it was assumed that there was a finite number of primes n.

---

[13] The Pythagoreans took a vow of silence with respect to the discovery of incommensurable numbers. However, one member of the society is said to have shared the secret result with others outside the sect. According to an apocryphal account, he was thrown into a lake for his betrayal and drowned.

5. Therefore, the assumption that there is a finite number of primes is false.
6. Therefore, there are an infinite number of primes.

Euclidean geometry included the parallel postulate or Euclid's fifth postulate. This postulate generated interest, as many mathematicians believed that it was unnecessary and could be proved as a theorem. It states as follows:

Definition 1.1 (*Parallel Postulate*) *If a line segment intersects two straight lines forming two interior angles on the same side that sum to less than two right angles, then the two lines, if extended indefinitely, meet on that side on which the angles sum to less than two right angles.*

This postulate was later proved to be independent of the other postulates with the development of non-Euclidean geometries in the nineteenth century. These include the *hyperbolic geometry* discovered independently by Bolyai and Lobachevsky, and *elliptic geometry* developed by Riemann. The standard model of Riemannian geometry is the sphere where lines are great circles.

The material in the Euclid's elements is a systematic development of geometry starting from the small set of axioms, postulates and definitions, leading to theorems logically derived from the axioms and postulates. Euclid's deductive method influenced later mathematicians and scientists. There are some jumps in reasoning, and the German mathematician, David Hilbert, later added extra axioms to address this.

The elements contains many well-known mathematical results such as Pythagoras's theorem, Thales theorem, sum of angles in a triangle, prime numbers, greatest common divisor and least common multiple, Euclidean algorithm, areas and volumes, tangents to a point and algebra.

The Euclidean algorithm is one of the oldest known algorithms and is employed to produce the greatest common divisor of two numbers. It is presented in the elements but was known well before Euclid. The algorithm to determine the GCD of two natural numbers, a and b, is given as follows:

1. Check if b is zero. If so, then a is the GCD.
2. Otherwise, the GCD (a, b) is given by GCD $(b, a \bmod b)$.

It is also possible to determine integers p and q such that $ap + bq = \text{GCD}(a, b)$.

The proof of the Euclidean algorithm is as follows. Suppose a and b are two positive numbers whose GCD is to be determined. Let r be the remainder when a is divided by b.

1. Clearly $a = qb + r$ where q is the quotient of the division.
2. Any common divisor of a and b is also a divisor or r (since $r = a - qb$).
3. Similarly, any common divisor of b and r will also divide a.
4. Therefore, the GCD of a and b is the same as the GCD of b and r.
5. The number r is smaller than b and we will reach $r = 0$ in many finite steps.
6. The process continues until $r = 0$.

Comment 1.1 *Algorithms are fundamental in computing as they define the procedure by which a problem is solved. A computer program implements the algorithm in some programming languages.*

Fig. 1.6 Eratosthenes'
measurement of the
circumference of the earth
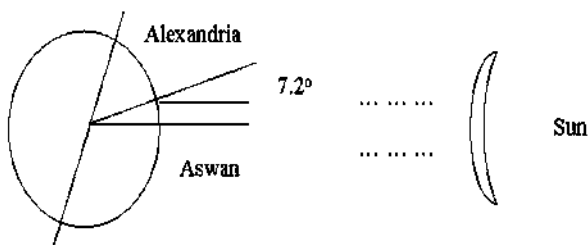
Eratosthenes was a Hellenistic mathematician and scientist who worked in the ancient library in Alexandria. This was the largest library in the ancient world. It was built during the Hellenistic period in the third century B.C. but destroyed by fire in A.D. 391.

Eratosthenes devised a system of latitude and longitude, and became the first person to estimate of the size of the circumference of the earth. His calculation proceeded as follows (Fig. 1.6):

1. On the summer solstice at noon in the town of Aswan[14] on the Tropic of Cancer in Egypt the sun appears directly overhead.
2. Eratosthenes believed that the earth was a sphere.
3. He assumed that rays of light from the sun came in parallel beams and reached the earth at the same time.
4. At the same time in Alexandria he had measured that the sun would be 7.2 south of the zenith.
5. He assumed that Alexandria was directly north of Aswan.
6. He concluded that the distance from Alexandria to Aswan was 7.2/360 of the circumference of the earth.
7. Distance between Alexandria and Aswan was 5,000 stadia (approximately 800 km).
8. He established a value of 252,000 stadia or approximately 39,600 km.

Eratosthenes's calculation was an impressive result for 200 B.C. The errors in his calculation were due to the following:

1. Aswan is not exactly on the Tropic of Cancer but it is actually 55 km north of it.
2. Alexandria is not exactly north of Aswan and there is a difference of 3 longitude.
3. The distance between Aswan and Alexandria is 729 km not 800 km.
4. Angles in antiquity could not be measured with a high degree of precision.
5. The angular distance is actually 7.08 and not 7.2.

Eratosthenes also calculated the approximate distance to the moon and sun and he also produced maps of the known world. He developed a very useful algorithm for

---

[14] The town of Aswan is famous today for the Aswan high dam, which was built in the 1960s. There was an older Aswan dam built by the British in the late nineteenth century. The new dam led to a rise in the water level of Lake Nasser and flooding of archaeological sites along the Nile. Several archaeological sites such as Abu Simbel and the temple of Philae were relocated to higher ground.

determining all of the prime numbers up to a specified integer. The method is known as the Sieve of Eratosthenes and the steps are as follows:

1. Write a list of the numbers from 2 to the largest number that you wish to test for primality. This first list is called A.
2. A second list B is created to list the primes. It is initially empty.
3. Number 2 is the first prime number and is added to the list of primes in B.
4. Strike off (or remove) 2 and all multiples of 2 from List A.
5. The first remaining number in List A is a prime number and this prime number is added to List B.
6. Strike off (or remove) this number and all multiples of this number from List A.
7. Repeat steps 5 through 7 until no more numbers are left in List A.

Comment 1.2 *The Sieve of Eratosthenes method is a well-known algorithm for determining prime numbers.*

Archimedes was a Hellenistic mathematician, astronomer and philosopher, and was born in Syracuse[15] in the third century B.C. He was a leading scientist in the Greco-Roman world, and he was credited with designing various innovative machines. He discovered the law of buoyancy known as Archimedes's principle:

> The buoyancy force is equal to the weight of the displaced fluid.

He is believed to have discovered the principle while sitting in his bath. He was so overwhelmed with his discovery that he rushed out onto the streets of Syracuse shouting "*Eureka*", to announce the discovery but forgot to put his clothes on.

The weight of the displaced liquid will be proportional to the volume of the displaced liquid. Therefore, if two objects have the same mass, the one with greater volume (or smaller density) has greater buoyancy. An object will float if its buoyancy force (i.e. the weight of liquid displaced) exceeds the downward force of gravity (i.e. its weight). If the object has exactly the same density as the liquid, then it will stay still, neither sinking nor floating upwards.

For example, a rock is generally a very dense material and will generally not displace its own weight. Therefore, a rock will sink to the bottom as the downward weight exceeds the buoyancy weight. However, if the weight of the object is less than that of the liquid it would displace, then it floats at a level where it displaces the same weight of the liquid that of the object.

Archimedes' inventions include the "*Archimedes Screw*" which was a screw pump that is still used today in pumping liquids and solids. Another of his inventions was the "*Archimedes Claw*", which was a weapon used to defend the city of Syracuse. It was also known as the "ship shaker" and it consisted of a crane arm from which a large metal hook was suspended. The claw would swing up and drop down on the attacking ship. It would then lift it out of the water and possibly sink it. Another of his inventions was said to be the "*Archimedes Heat Ray*". This device is said to have consisted of a number of mirrors that allowed sunlight to be focused on an enemy ship thereby causing it to go on fire (Fig. 1.7).

---

[15] Sysacuse is located on the island of Sicily in southern Italy.

Fig. 1.7 "Archimedes in thought" by Fetti

Archimedes' made good contributions to mathematics including developing a good approximation to π, as well as contributions to the positional numbering system, geometric series and to maths physics. He also solved several interesting problems, e.g. the calculation of the composition of cattle in the herd of the Sun god by solving a number of simultaneous Diophantine equations. The herd consisted of bulls and cows with one part of the herd consisting of white, the second part black, the third spotted and the fourth brown. Various constraints were then expressed in Diophantine equations. The problem was to determine the precise composition of the herd. Diophantine equations are named after Diophantus who worked on the number theory in the third century B.C.

There is a well-known anecdote concerning Archimedes and the crown of King Hiero II. The king wished to determine whether his new crown was made entirely of solid gold, and that the goldsmith had added no substitute silver. Archimedes was required to solve the problem without damaging the crown, and as he was taking a bath he realized that if the crown was placed in water, the displaced water would give him the volume of the crown. From this he could then determine the density of the crown and therefore whether it consisted entirely of gold.

Archimedes also calculated an upper bound of the number of grains of sands in the known universe. The largest number in common use at the time was a myriad myriad (100 million), where a myriad is 10,000. Archimedes' numbering system went up to 8×10^{16} and he also developed the laws of exponents, i.e. $10^a 10^b = 10^{a+b}$. His calculation of the upper bound included not only the grains of sand on each beach, but on the earth filled with sand and the known universe filled with sand. His final estimate of the upper bound for the number of grains of sand in a filled universe was 10^{64}.

It is possible that he may have developed the odometer.[16] This instrument could calculate the total distance travelled on a journey and was described by the Roman

---

[16] The origin of the word "odometer" is from the Greek words 'οδος (meaning journey) and μετρον meaning (measure).

engineer Vitruvius around 25 B.C. It employed a wheel with a diameter of 4 ft that turned 400 times in every mile.[17] The device included gears and pebbles and a 400-tooth cogwheel that turned once every mile and caused one pebble to drop into a box. The total distance travelled was determined by counting the pebbles in the box.

Aristotle was born in Macedonia and became a student of Plato in Athens. Plato had founded a school (known as Plato's academy) in Athens in the fourth century B.C., and this school remained open until 529 A.D. Aristotle founded his own school (known as the Lyceum) in Athens. He was also the tutor of Alexander the Great. He made contributions to physics, biology, logic, politics, ethics and metaphysics.

Aristotle's starting point to the acquisition of knowledge was the senses, and he believed that these were essential to acquire knowledge. This position is the opposite from Plato who argued that the senses deceive and should not be relied upon. Plato's writings are mainly written in dialogues involving his former mentor Socrates (Fig. 1.8).[18]

---

[17] The figures given here are for the distance of one Roman mile. This is less than a standard mile in the Imperial System.

[18] Socrates was a moral philosopher who deeply influenced Plato. His method of enquiry into philosophical problems and ethics was by questioning. Socrates himself maintained that he knew

Table 1.1 Syllogisms, relationship between terms

Relationship	Abbreviations
Universal affirmation	A
Universal negation	E
Particular affirmation	I
Particular negation	O

Aristotle made important contributions to formal reasoning with his development of *syllogistic logic*. His collected works on logic is called the *Organon* and it was used in his school in Athens. Syllogistic logic (also known as term logic) consists of reasoning with two premises and one conclusion. Each premise consists of two terms and a common middle term. The conclusion links the two unrelated terms from the premises. For example:

Premise 1   All Greeks are Mortal

Premise 2   Socrates is a Greek.

. . . .

Conclusion   Socrates is Mortal

The common middle term is "Greek", which appears in the two premises. The two unrelated terms from the premises are "Socrates" and "Mortal". The relationship between the terms in the first premise is that of the *universal*, i.e. anything or any person that is a Greek is mortal. The relationship between the terms in the second premise is that of the *particular*, i.e. Socrates is a person that is a Greek. The conclusion from the two premises is that Socrates is mortal, i.e. a particular relationship between the two unrelated terms "Socrates" and "Mortal".

The syllogism above is a valid syllogistic argument. Aristotle studied the various possible syllogistic arguments and determined those that were valid and invalid. There are several candidate relationships that may exist between the terms in a premise, and these are defined in Table 1.1. In general, a syllogistic argument will be of the form:

$$S \; x \; M$$

$$M \; y \; P$$

. . . . . .

$$S \; z \; P$$

nothing (Socratic ignorance). However, from his questioning it became apparent that those who thought they were clever were not really that clever after all. His approach obviously would not have made him very popular with the citizens of Athens. Socrates had consulted the oracle at Delphi to find out who was the wisest of all men, and he was informed that there was no one wiser than him. Socrates was sentenced to death for allegedly corrupting the youth of Athens, and the sentence was carried out by Socrates being forced to take hemlock (a type of poison). The juice of the hemlock plant was prepared for Socrates to drink.

where *x*, *y* and *z* may be universal affirmation, universal negation, particular affirmation and particular negation. Syllogistic logic is described in more detail in [ORg:06]. Aristotle's work was highly regarded in classical and medieval times and the philosopher, Kant, believed that there was nothing else to invent in logic. There was an alternate system of logic proposed by the Stoics in Hellenistic times, i.e. an early form of propositional logic that was developed by Chrysippus[19] in the third century B.C. Aristotelian logic is mainly of historical interest today.

Aquinas,[20] a thirteenth century Christian theologian and philosopher, was deeply influenced by Aristotle, and referred to him as "the philosopher". Aquinas was an empiricist (i.e. he believed that all knowledge was gained by sense experience), and he used some of Aristotle's arguments to offer five proofs of the existence of God. These arguments included the *Cosmological argument* and the *Design argument*. The Cosmological argument used Aristotle's ideas on the scientific method and causation. Aquinas argued that there is a first cause and he deduced that this first cause is God.

1. Every effect has a cause
2. Nothing can cause itself
3. A causal chain cannot be of infinite length
4. Therefore there must be a first cause

The Antikythera [Pri:59] was an ancient mechanical device that is believed to have been designed to calculate astronomical positions. It was discovered in 1902 in a wreck off the Greek island of Antikythera, and dates from about 80 B.C. It is one of the oldest known geared devices, and believed to have been used for calculating the position of the sun, moon, stars and planets for a particular date entered.

The Romans appear to have been aware of a device similar to Antikythera that was capable of calculating the position of the planets. The island of Antikythera was well known in the Greek and Roman period for its displays of mechanical engineering.

1.5 The Romans

Rome is said to have been founded[21] by Romulus and Remus about 750 B.C. Early Rome covered a small part of Italy but it gradually expanded in size and importance. The Romans destroyed Carthage[22] in 146 B.C. to become the major power in the

---

[19] Chrysippus was the head of the Stoics in the third century B.C.

[20] Aquinus's (or St. Thomas's) most famous work is *Sumna Theologicae*.

[21] The *Aenid* by Virgil suggests that the Romans were descended from survivors of the Trojan War, and that Aeneas brought surviving Trojans to Rome after the fall of Troy.

[22] Carthage was located in Tunisia, and the wars between Rome and Carthage are known as the Punic wars. Hannibal was one of the great Carthaginan military commanders, and during the second Punic war, he brought his army to Spain, marched through Spain and crossed the Pyrnees. He then marched along southern France and crossed the Alps into Northern Italy. His army also consisted of war elephants. Rome finally defeated Carthage and destroyed the city.

Fig. 1.9 Julius Caesar

Fig. 1.10 Roman numbers

I = 1
V = 5
X = 10
L = 50
C = 100
D = 500
M = 1000

Mediterranean. The Romans colonised the Hellenistic world, and they were influenced by Greek culture and mathematics. Julius Caesar conquered the Gauls in 58 B.C. (Fig. 1.9).

The Gauls consisted of several disunited Celtic[23] tribes. Vercingetorix succeeded in uniting them, but he was defeated by at the siege of Alesia in 52 B.C. (Fig. 1.10).

The Roman number system uses letters to represent numbers and a number consists of a sequence of letters. The evaluation rules specify that if a number follows a smaller number then the smaller number is subtracted from the larger number, e.g. IX represents 9 and XL represents 40. Similarly, if a smaller number followed a larger number they were generally added, e.g. MCC represents 1,200. They had no zero in their system.

---

[23] The Celtic period commenced around 1000 B.C. in Hallstaat (near Salzburg in Austria). The Celts were skilled in working with iron and bronze, and they gradually expanded into Europe. They eventually reached Britain and Ireland around 600 B.C. The early Celtic period was known as the 'Hallstaat period' and the later Celtic period is known as 'La Téne'. The later La Téne period is characterised by the quality of ornamentation produced. The Celtic museum in Hallein in Austria provides valuable information and artefacts on the Celtic period. The Celtic language would have similarities to the Irish language. However, the Celts did not employ writing, and the Ogham writing used in Ireland was developed in the early Christian period.

Fig. 1.11 Caesar Cipher

Alphabet Symbol	abcde fghij klmno pqrst uvwxyz
Cipher Symbol	dfegh ijklm nopqr stuvw xyzabc

The use of Roman numerals was cumbersome in calculation, and an abacus was often employed. An abacus is a device that is usually of wood and has a frame that holds rods with freely sliding beads mounted on them. It is used as a tool to assist calculation, and is useful for keeping track of the sums and the carries of calculations.

The abascus consisted of several columns in which beads or pebbles were placed. Each column represented powers of 10, i.e. 10^0, 10^1, 10^2, 10^3, etc. The column to the far right represented 1, the column to the left 10, next column to the left 100 and so on. Pebbles[24] (calculi) were placed in the columns to represent different numbers, e.g. the number represented by an abacus with four pebbles on the far right, two pebbles in the column to the left, and three pebbles in the next column to the left is 324. The calculation was performed by moving pebbles from column to column.

Merchants introduced a set of weights and measures (including the *libra* for weights and the *pes* for lengths). They developed an early banking system to provide loans for business, and commenced minting coins about 290 B.C. The Romans also made contributions to calendars, and Julius Caesar introduced the Julian calendar in 45 B.C. It has a regular year of 365 days divided into 12 months and a leap day is added to February every 4 years. It remained in use up to the twentieth century, but has since been replaced by the Gregorian calendar. The problem with the Julian calendar is that too many leap years are added over time. The Gregorian calendar was first introduced in 1582.

Caesar employed a substitution cipher on his military campaigns to enable important messages to be communicated safely. The cipher involved the substitution of each letter in the plaintext (i.e. the original message) by a letter a fixed number of positions down in the alphabet. For example, a shift of three positions causes the letter B to be replaced by E, the letter C by F, and so on. The cipher is easily broken, as the frequency distribution of letters may be employed to determine the mapping. The cipher is defined as shown in (Fig. 1.11).

The process of enciphering a message (i.e. plaintext) involves looking up each letter in the plaintext and writing down the corresponding cipher letter. The decryption involves the reverse operation, i.e. for each cipher letter the corresponding plaintext letter is identified from the table.

The encryption may also be represented using modular arithmetic,[25] with the numbers 0–25 representing the alphabet letters, and addition (modulo 26) is used to perform the encryption.

---

[24] The origin of the word "Calculus" is from Latin and means a small stone or pebble used for counting.

[25] Modular arithmetic is discussed in Chap. 7.

The emperor Augustus[26] employed a similar substitution cipher (with a shift key of 1). The Caesar cipher remained in use up to the early twentieth century. However, by then, frequency analysis techniques were available to break the cipher. The Romans employed the mathematics that had been developed by the Greeks rather than making fundamental contributions.

1.6   Islamic Influence

Islamic mathematics refers to mathematics developed in the Islamic world from the birth of Islam in the early seventh century up until the seventeenth century. The Islamic world commenced with the prophet Mohammed in Mecca, and spread throughout the Middle East, North Africa and Spain. Islamic scholars translated the works of the Greeks into Arabic, and this led to the preservation of the Greek texts during the Dark Ages in Europe. The Islamic scholars developed the existing mathematics further.

The Moors[27] invaded Spain in the A.D. eighth century, and they ruled large parts of the Iberian Peninsula for several centuries. The Moorish influence[28] in Spain continued until the time of the Catholic Monarchs[29] in the fifteenth century. Ferdinand and Isabella united Spain, defeated the Moors, and expelled them from the country.

Islamic mathematicians and scholars were based in several countries including the Middle East, North Africa and Spain. Early work commenced in Baghdad, and the mathematicians were influenced by the work of Hindu mathematicians who had introduced the decimal system and decimal numerals. Al Khwarizmi[30] adopted this system in the ninth century, and the resulting system is known as the *Hindu–Arabic number system*.

Many caliphs were enlightened rulers and encouraged scholarship in mathematics and science. This led to the translation of the existing Greek texts, and a centre of translation and research was set up in Baghdad leading to the translation of the works of Euclid, Archimedes, Apollonius and Diophantus. Al-Khwarizmi made

---

[26] Augustus was the first Roman emperor whose reign ushered in a period of peace and stability following the bitter civil wars. He was the adopted son of Julius Caesar and was called Octavion before he became emperor. The earlier civil wars were between Caesar and Pompey, and following Caesar's assassination a civil war broke out between Mark Anthony and Octavion. Octavion defeated Anthony and Cleopatra at the battle of Actium.

[27] The origin of the word "Moor" is from the Greek work μνορος meaning very dark. It referred to the fact that many of the original Moors who came to Spain were from Egypt, Tunisia and other parts of North Africa.

[28] The Moorish influence includes the construction of various castles (*alcazar*), fortresses (*alcalzaba*) and mosques. One of the most striking Islamic sites in Spain is the palace of Alhambra in Granada, and this site represents the zenith of Islamic art.

[29] .The Catholic Monarchs refer to Ferdinand of Aragon and Isabella of Castille who married in 1469. They captured Granada (the last remaining part of Spain controlled by the Moors) in 1492.

[30] The origin of the word "algorithm" is from the name of the Islamic scholar Al-Khwarizmi.

contributions to early classical algebra, and the word algebra comes from the Arabic word "*al jabr*" that appears in a textbook by Al-Khwarizmi.

The Islamic contribution to algebra was an advance on the achievements of the Greeks. They developed a broader theory that treated rational and irrational numbers as algebraic objects, and moved away from the Greek concept of mathematics as being essentially Geometry. Later Islamic scholars applied algebra to arithmetic and geometry. This included contributions to reduce geometric problems such as duplicating the cube to algebraic problems. Eventually in the fifteenth century this led to the use of symbols such as:

$$x^n \cdot x^m = x^{m+n}$$

The poet, Omar Khayyam, was also a mathematician.[31] He did work on the classification of cubic equations with geometric solutions. Others applied algebra to geometry, and aimed to study curves by using equations. Other scholars made contributions to the theory of numbers, e.g. a theorem that allows pairs of amicable numbers to be found. Amicable numbers are two numbers such that each is the sum of the proper divisors of the other. They were aware of Wilson's theory in number theory i.e. if p is a prime number then p divides $(p - 1)! + 1$.

Moorish Spain became a centre of learning with Islamic and other scholars coming to study at its universities. Many texts on Islamic mathematics were translated from Arabic into Latin, these were invaluable in the renaissance in European learning and mathematics from the thirteeenth century.

1.7   Chinese and Indian Mathematics

The development of mathematics commenced in China about 1000 B.C. and was independent of developments in other countries. The emphasis was on problem solving rather than on conducting formal proofs. This involved finding the solution to practical problems such as the calendar, the prediction of the positions of the heavenly bodies, land measurement, conducting trade, and the calculation of taxes.

The Chinese employed counting boards as mechanical aids for calculation from the fourth century B.C. These are similar to abaci and are usually made of wood or metal, and contained carved grooves between which beads, pebbles or metal discs were moved.

Early Chinese mathematics was written on bamboo strips and included work on arithmetic and astronomy. The Chinese method of learning and calculation in mathematics was learning by analogy. This involves a person acquiring knowledge from observation of how a problem is solved, and then applying this knowledge for problem solving to similar kinds of problems.

The Chinese had their version of Pythagoras's Theorem and applied it to practical problems. They were familiar with the Chinese remainder theorem, the formula for

---

[31] I am aware of no other mathematician who was also a poet.

finding the area of a triangle, as well as showing how polynomial equations (up to degree ten) could be solved. They showed how geometric problems could be solved by algebra, how roots of polynomials could be solved, how quadratic and simultaneous equations could be solved, and how the area of various geometric shapes such as rectangles, trapezia and circles could be computed. Chinese mathematicians were familiar with the formula to calculate the volume of a sphere. The best approximation that the Chinese had to π was 3.14159, and this was obtained by approximations from inscribing regular polygons with 3×2^n sides in a circle.

The Chinese made contributions to number theory including the summation of arithmetic series and solving simultaneous congruences. The Chinese remainder theorem deals with finding the solutions to a set of simultaneous congruences in modular arithmetic. Chinese astronomers made accurate observations, which were used to produce a new calendar in the sixth century. This was known as the Taming Calendar and was based on a cycle of 391 years.

Indian mathematicians have made important contributions such as the development of the decimal notation for numbers that is now used throughout the world. This was developed in India sometime between 400 B.C. and A.D. 400 Indian mathematicians also invented zero and negative numbers, and also did early work on the trigonometric functions of sine and cosine. The knowledge of the decimal numerals reached Europe through Arabic mathematicians, and the resulting system is known as the Hindu–Arabic numeral system.

The Sulva Sutras is a Hindu text that documents Indian mathematics and dates from about 400 B.C. The Sutras were familiar with the statement and proof of Pythagoras's theorem, Rational numbers, quadratic equations, as well as the calculation of the square root of 2 to five decimal places.

1.8 Review Questions

1. Discuss the strengths and weaknesses of the various numbering system.
2. Describe the ciphers used during the Roman civilisation and write a program to implement one of these.
3. Discuss the nature of an algorithm and its importance in computing.
4. Discuss the working of an abacus and its application to calculation.
5. What are the differences between syllogistic logic and propositional and predicate logic?

1.9 Summary

Software is pervasive throughout society and has transformed the world in which we live in. New technology has led to improvements in all aspects of our lives including medicine, transport, education, and so on. The pace of change of new technology

is relentless, with new versions of technology products becoming available several times a year.

This chapter considered some of the contributions of early civilisations to computing. We commenced our journey with an examination of some of the contributions of the Babylonians. We then moved forward to consider some of the achievements of the Egyptians, the Greek and Romans, Islamic scholars, and the Indians and Chinese.

The Babylonians developed a base 60 number system, and recorded their mathematical knowledge on clay cuneiform tablets. These tablets included tables for multiplication, division, squares, and square roots and the calculation of area. They were familiar with techniques that allowed the solution of a linear equation and one root of a quadratic equation to be determined.

The Egyptian civilization developed along the River Nile and lasted over 3,000 years. They applied their knowledge of mathematics to solve practical problem such as measuring the annual Nile flooding, and constructing temples and pyramids.

The Greeks and the later Hellenistic period made important contributions to western civilisation. This included contributions to philosophy, architecture, politics, logic, geometry and mathematics. The Euclidean algorithm is used to determine the greatest common divisor of two numbers. Eratosthenes developed an algorithm to determine the prime numbers up to a given number. Archimedes invented the "Archimedes Screw", the "Archimedes Claw", and a type of heat ray.

The Islamic civilisation helped to preserve western knowledge that was lost during the dark ages in Europe, and they also continued to develop mathematics and algebra. Hindu mathematicians introduced the decimal notation that is familiar today. Islamic mathematicians adopted it and the resulting system is known as the Hindu–Arabaic system.

Chapter 2
Sets, Relations and Functions

Key Topics

Sets
Set Operations
Russell's Paradox
Relations
Composition of Relations
Reflexive, Symmetric and Transitive Relations
Functions
Partial and Total Functions
Injective, Surjective and Transitive Functions

2.1 Introduction

This chapter provides an introduction to the fundamental building blocks in mathematics such as sets, relations and functions. Sets are collections of well-defined objects, relations indicate relationships between members of two sets A and B and functions are a special type of relation where there is exactly or at most[1] one relationship for each element $a \in A$ with an element in B.

A set is a collection of well-defined objects that contains no duplicates. The term "well defined" means that for a given value it is possible to determine whether or not it is a member of the set. There are many examples of sets such as the set of natural numbers $\mathbb{N}$, the set of integer numbers $\mathbb{Z}$ and the set of rational numbers $\mathbb{Q}$. The set of natural numbers $\mathbb{N}$ is an infinite set consisting of the numbers $\{1, 2, \dots\}$. Venn diagrams may be used to represent sets pictorially.

A binary relation $R(A, B)$ where A and B are sets is a subset of the Cartesian product $(A \times B)$ of A and B. The domain of the relation is A and the co-domain of the relation is B. The notation aRb signifies that there is a relation between a and b and that

---

[1] We distinguish between total and partial functions. A total function $f : A \to B$ is defined for every element in A whereas a partial function may be undefined for one or more values in A.

$(a, b) \in R$. An n-ary relation $R(A_1, A_2, \ldots, A_n)$ is a subset of $(A_1 \times A_2 \times \cdots \times A_n)$. However, an n-ary relation may also be regarded as a binary relation $R(A, B)$ with $A = A_1 \times A_2 \times \cdots \times A_{n-1}$ and $B = A_n$.

Functions may be total or partial. A total function $f : A \rightarrow B$ is a special relation such that for each element $a \in A$ there is exactly one element $b \in B$. This is written as $f(a) = b$. A partial function differs from a total function in that the function may be undefined for one or more values of A. The domain of a function (denoted by **dom** f) is the set of values in A for which the function is defined. The domain of the function is A provided that f is a total function. The co-domain of the function is B.

2.2 Set Theory

A set is a fundamental building block in mathematics, and it is defined as a collection of well-defined objects. The elements in a set are of the same kind, and they are distinct with no repetition of the same element in the set[2]. Most sets encountered in computer science are finite as computers can only deal with finite entities. Venn diagrams[3] are often employed to give a pictorial representation of a set and may be used to illustrate various set operations such as set union, intersection and set difference.

There are many well-known examples of sets including the set of natural numbers denoted by $\mathbb{N}$, the set of integers denoted by $\mathbb{Z}$, the set of rational numbers is denoted by $\mathbb{Q}$, the set of real numbers denoted by $\mathbb{R}$ and the set of complex numbers denoted by $\mathbb{C}$.

Example 2.1 The following are examples of sets:

- The books on the shelves in a library.
- The books currently overdue from the library.
- The customers of a bank.
- The bank accounts in a bank.
- The set of natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$.
- The integer numbers $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.
- The non-negative integers $\mathbb{Z}^+ = \{0, 1, 2, 3, \ldots\}$.
- The set of prime numbers $= \{2, 3, 5, 7, 11, 13, 17, \ldots\}$.
- The rational numbers is the set of quotients of integers

$$\mathbb{Q} = \{p/q : p, q \in \mathbb{Z} \text{ and } q \neq 0\}.$$

A finite set may be defined by listing all of its elements. For example, the set $A = \{2, 4, 6, 8, 10\}$ is the set of all even natural numbers less than or equal to 10. The

---

[2] There are mathematical objects known as multi-sets or bags that allow duplication of elements. For example, a bag of marbles may contain three green marbles, two blue and one red marble.

[3] The British logician, John Venn, invented the Venn diagram. It provides a visual representation of a set and the various set theoretical operations. Their use is limited to the representation of two or three sets as they become cumbersome with a larger number of sets.

order in which the elements are listed is not relevant: i.e., the set $\{2, 4, 6, 8, 10\}$ is the same as the set $\{8, 4, 2, 10, 6\}$.

Sets may be defined by using a predicate to constrain set membership. For example, the set $S = \{n : \mathbb{N} : n \leq 10 \wedge n \bmod 2 = 0\}$ also represents the set $\{2, 4, 6, 8, 10\}$. That is, the use of a predicate allows a new set to be created from an existing set by using the predicate to restrict membership of the set. The set of even natural numbers may be defined by a predicate over the set of natural numbers that restricts membership to the even numbers. It is defined by:

$$\text{Evens} = \{x | x \in \mathbb{N} \wedge even(x)\}.$$

In this example, $even(x)$ is a predicate that is true if x is even and false otherwise. In general, $A = \{x \in E | P(x)\}$ denotes a set A formed from a set E using the predicate P to restrict membership of A to those elements of E for which the predicate is true.

The elements of a finite set S are denoted by $\{x_1, x_2, \ldots, x_n\}$. The expression $x \in S$ denotes set membership and indicates that the element x is a member of the set S. The expression $x \notin S$ indicates that x is not a member of the set S.

A set S is a subset of a set T (denoted $S \subseteq T$) if whenever $s \in S$ then $s \in T$, and in this case the set T is said to be a superset of S (denoted $T \supseteq S$). Two sets S and T are said to be equal if they contain identical elements: i.e., $S = T$ if and only if $S \subseteq T$ and $T \subseteq S$. A set S is a proper subset of a set T (denoted $S \subset T$) if $S \subseteq T$ and $S \neq T$. That is, every element of S is an element of T and there is at least one element in T that is not an element of S. In this case, T is a proper superset of S (denoted $T \supset S$).
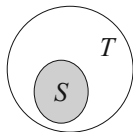
The empty set (denoted by $\emptyset$ or $\{\}$) represents the set that has no elements. Clearly $\emptyset$ is a subset of every set. The singleton set containing just one element x is denoted by $\{x\}$, and clearly $x \in \{x\}$ and $x \neq \{x\}$. Clearly, $y \in \{x\}$ if and only if $x = y$.

Example 2.2

(i) $\{1, 2\} \subseteq \{1, 2, 3\}$.
(ii) $\emptyset \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

The cardinality (or size) of a finite set S defines the number of elements present in the set. It is denoted by $|S|$. The cardinality of an infinite[4] set S is written as $|S| = \infty$.

---

[4] The natural numbers, integers and rational numbers are countable sets whereas the real and complex numbers are uncountable sets.

Example 2.3

(i)   Given $A = \{2, 4, 5, 8, 10\}$ then $|A| = 5$.
(ii) Given $A = \{x \in \mathbb{Z} : x^2 = 9\}$ then $|A| = 2$.
(iii) Given $A = \{x \in \mathbb{Z} : x^2 = -9\}$ then $|A| = 0$.

2.2.1 Set Theoretical Operations

Several set theoretical operations are considered in this section. These include the
Cartesian product operation, the set union operation, the set intersection operation,
the set difference operation and the symmetric difference operation.

Cartesian Product The Cartesian product allows a new set to be created from
existing sets. The Cartesian[5] product of two sets S and T (denoted $S \times T$) is the set
of ordered pairs $\{(s, t) | s \in S, t \in T\}$. Clearly, $S \times T \neq T \times S$ and so the Cartesian
product of two sets is not commutative. Two ordered pairs (s_1, t_1) and (s_2, t_2) are
considered equal if and only if $s_1 = s_2$ and $t_1 = t_2$.

The Cartesian product may be extended to that of n sets $S_1, S_2, \ldots, S_n$. The Carte-
sian product $S_1 \times S_2 \times \cdots \times S_n$ is the set of ordered tuples $\{(s_1, s_2, \ldots, s_n) | s_1 \in S_1,
s_2 \in S_2, \ldots, s_n \in S_n\}$. Two ordered n-tuples $(s_1, s_2, \ldots, s_n)$ and $(s_1', s_2', \ldots, s_n')$ are
considered equal if and only if $s_1 = s_1', s_2 = s_2', \ldots, s_n = s_n'$.

The Cartesian product may also be applied to a single set S to create ordered
n-tuples of S: i.e., $S^n = S \times S \times \cdots \times S$ (n times).

Power Set The power set of a set A (denoted $\mathbb{P}A$) denotes the set of all subsets of A.
For example, the power set of the set $A = \{1, 2, 3\}$ has eight elements and is given
by:

$$\mathbb{P}A = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

There are $2^3 = 8$ elements in the power set of $A = \{1, 2, 3\}$ and the cardinality of A
is 3. In general, there are $2^{|A|}$ elements in the power set of A.

Theorem 2.1 (Cardinality of Power Set of A) There are $2^{|A|}$ elements in the power
set of A.

Proof Let $|A| = n$ then the subsets of A include subsets of size $0, 1, \ldots, n$. There
are $\binom{n}{k}$ subsets of A of size k. Therefore, the total number of subsets of A is the total
number of subsets of size $0, 1, 2, \ldots$ up to n. That is,

$$|\mathbb{P}A| = \sum_{k=0}^{n} \binom{n}{k}.$$

---

[5] Cartesian product is named after René Descartes who was a famous 17th French mathematician
and philosopher. He invented the Cartesian coordinates system that links geometry and algebra,
and allows geometric shapes to be defined by algebraic equations.

The binomial theorem states that:

$$(1 + x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k.$$

Therefore, putting $x = 1$ we get that:

$$2^n = (1 + 1)^n = \sum_{k=0}^{n} \binom{n}{k} 1^k = |\mathbb{P}A|.$$

Union and Intersection Operations   The union of two sets A and B is denoted by $A \cup B$. It results in a set that contains all of the members of A and of B and is defined by:

$$A \cup B = \{r \,|\, r \in A \text{ or } r \in B\}.$$

For example, suppose $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$ then $A \cup B = \{1, 2, 3, 4\}$. Set union is a commutative operation: i.e., $A \cup B = B \cup A$. Venn Diagrams are used to illustrate these operations pictorially.
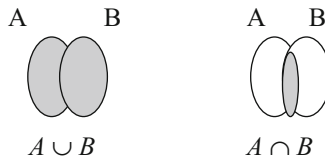
The intersection of two sets A and B is denoted by $A \cap B$. It results in a set containing the elements that A and B have in common and is defined by:

$$A \cap B = \{r \,|\, r \in A \text{ and } r \in B\}.$$

Suppose $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$ then $A \cap B = \{2, 3\}$. Set intersection is a commutative operation: i.e., $A \cap B = B \cap A$.

Union and intersection are binary operations but may be extended to more generalized union and intersection operations. For example,

$$\bigcup_{i=1}^{n} A_i \text{ denotes the union of } n \text{ sets,}$$

$$\bigcap_{i=1}^{n} A_i \text{ denotes the intersection of } n \text{ sets.}$$

Set Difference Operations   The set difference operation $A \backslash B$ yields the elements in A that are not in B. It is defined by:

$$A \backslash B = \{a \,|\, a \in A \text{ and } a \notin B\}.$$

For A and B defined as $A = \{1, 2\}$ and $B = \{2, 3\}$ we have $A \backslash B = \{1\}$ and $B \backslash A = \{3\}$. Clearly, set difference is not commutative: i.e., $A \backslash B \neq B \backslash A$. Clearly, $A \backslash A = \emptyset$ and $A \backslash \emptyset = A$.

The symmetric difference of two sets A and B is denoted by $A \bigtriangleup B$ and is defined by:

$$A \bigtriangleup B = A \backslash B \cup B \backslash A.$$

The symmetric difference operation is commutative: i.e., $A \bigtriangleup B = B \bigtriangleup A$. Venn diagrams are used to illustrate these operations pictorially.
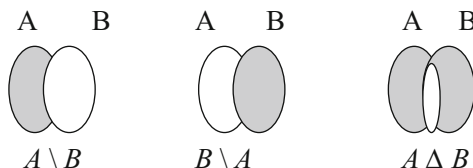
The complement of a set A (with respect to the universal set U) is the elements in the universal set that are not in A. It is denoted by A^c (or A') and is defined as:

$$A^c = \{u | u \in U \text{ and } u \notin A\} = U \backslash A.$$

The complement of the set A is illustrated by the shaded area below:
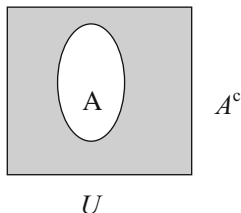
2.2.2   Properties of Set Theoretical Operations

The set union and set intersection properties are commutative and associative. The properties are listed in Table 2.1.

We give a proof of the distributive property.

Proof of Properties (Distributive Property) To show $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

Suppose $x \in A \cap (B \cup C)$ then

$$x \in A \wedge x \in (B \cup C),$$
$$\Rightarrow x \in A \wedge (x \in B \vee x \in C),$$

Table 2.1 Properties of set operations

Property	Description
Commutative	Union and intersection operations are commutative: i.e., $S \cup T = T \cup S$ $S \cap T = T \cap S$
Associative	Union and intersection operations are associative: i.e., $R \cup (S \cup T) = (R \cup S) \cup T$ $R \cap (S \cap T) = (R \cap S) \cap T$
Identity	The identity under set union is Ø and the identity under intersection is U. $S \cup \varnothing = \varnothing \cup S = S$ $S \cap U = U \cap S = S$
Distributive	The union operator distributes over the intersection operator and vice versa. $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$ $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$
DeMorgan's[a] law	The complement of $S \cup T$ is given by: $(S \cup T)^c = S^c \cap T^c$. The complement of $S \cap T$ is given by: $(S \cap T)^c = S^c \cup T^c$

[a]De Morgan's law is named after Augustus De Morgan, a nineteenth century English mathematician who was a contemporary of George Boole.

$$\Rightarrow (x \in A \land x \in B) \lor (x \in A \land x \in C),$$

$$\Rightarrow x \in (A \cap B) \lor x \in (A \cap C),$$

$$\Rightarrow x \in (A \cap B) \cup (A \cap C).$$

Therefore, $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$.
   Similarly, $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$.
   Therefore, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

2.2.3   Russell's Paradox

Bertrand Russell was a famous British logician, mathematician and philosopher. He was the co-author with Alfred Whitehead of *Principia Mathematica*, which aimed to derive all of the truths of mathematics from logic. Russell's paradox was discovered by Bertrand Russell in 1901, and showed that the system of set theory being proposed by Frege contained a contradiction (Fig. 2.1).

Question 2.1 (Posed by Russell to Frege) Is the set of all sets that do not contain themselves as members a set?

Russell's Paradox   Let $A = \{S$ a set and $S \notin S\}$. Is $A \in A$? Then $A \in A \Rightarrow A \notin A$ and vice versa. Therefore, a contradiction arises in either case and there is no such set A.

Two ways of avoiding the paradox were developed in 1908, and these were Russell's theory of types and Zermelo set theory. Russell's theory of types was a response to the paradox that argued that the set of all sets is ill formed. Russell developed a hierarchy of sets with individual elements at the lowest level, sets of elements at the next level, sets of sets of elements at the next level and so on. It is then prohibited for a set to contain members of different types.

A set of elements has a different type from its elements, and one cannot speak of the set of all sets that do not contain themselves as members as these are of different types. The other way of avoiding the paradox was Zermelo's axiomatization of set theory.

Remark   Russell's paradox may also be illustrated by the story of a town that has exactly one barber who is male. *The barber shaves all and only those men in town who do not shave themselves*. The question is who shaves the barber.

If the barber does not shave himself then according to the rule he is shaved by the barber (i.e., himself). If he shaves himself then according to the rule he is not shaved by the barber (i.e., himself).

The paradox occurs due to self-reference in the statement and a logical examination shows that the statement is a contradiction.

2.3   Relations

A binary relation $R(A, B)$ where A and B are sets is a subset of $A \times B$: i.e., $R \subseteq A \times B$. The notation aRb signifies that $(a, b) \in R$.

A binary relation $R(A, A)$ is a relation between A and A. This type of relation may always be composed with itself, and its inverse is also a binary relation on A. The identity relation on A is defined by $a\, i_A a$ for all $a \in A$.

Example 2.4 There are many examples of relations:

(i)   The relation tall defined on a set of students in a class where $(a, b) \in R$ if the height of a is greater than the height of b.
(ii) The relation between A and B where $A = \{0, 1, 2\}$ and $B = \{3, 4, 5\}$ with R given by:

$$R = \{(0, 3), (0, 4), (1, 4)\}.$$

(iii) The relation less than $(<)$ between $\mathbb{R}$ and $\mathbb{R}$ is given by:

$$\{(x, y) \in \mathbb{R}^2 : x < y\}.$$

(iv) A bank may represent the relationship between the set of accounts and the set of customers by a relation. The implementation of a bank account could be a positive integer with at most eight decimal digits.

The relationship between accounts and customers may be done with a relation $R \subseteq A \times B$, with the set A chosen to be the set of natural numbers, and the set B chosen to be the set of all human beings alive or dead. The set A could also be chosen to be $A = \{n \in \mathbb{N} : n < 10^8\}$.

A relation $R(A, B)$ may be represented pictorially. This is referred to as the graph of the relation and is illustrated in the diagram below. An arrow from x to y is drawn if (x, y) is in the relation. Thus for the height relation R given by $\{(a, p), (a, r), (b, q)\}$ an arrow is drawn from a to p, from a to r and from b to q to indicate that (a, p), (a, r) and (b, q) are in the relation R.
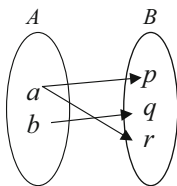
The pictorial representation of the relation makes it easy to see that the height of a is greater than the height of p and r, and that the height of b is greater than the height of q.

An *n*-ary relation $R\,(A_1, A_2, \ldots, A_n)$ is a subset of $(A_1 \times A_2 \times \cdots \times A_n)$. However, an *n*-ary relation may also be regarded as a binary relation $R(A, B)$ with $A = A_1 \times A_2 \times \cdots \times A_{n-1}$ and $B = A_n$.

Fig. 2.2 Reflexive relation
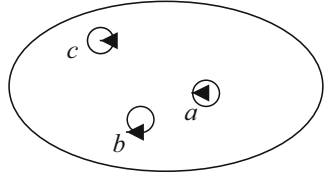
Fig. 2.3 Symmetric relation
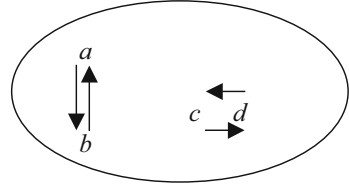
2.3.1 Reflexive, Symmetric and Transitive Relations

There are various types of relations on a set A including reflexive, symmetric and transitive relations.

(i)   A relation on a set A is *reflexive* if $(a, a) \in R$ for all $a \in A$.
(ii)   A relation R is *symmetric* if whenever $(a, b) \in R$ then $(b, a) \in R$.
(iii) A relation is *transitive* if whenever $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$.

A relation that is reflexive, symmetric and transitive is termed an *equivalence relation*. An equivalence relation gives a partition of the set A.

Example 2.5 (Reflexive Relation) A relation is reflexive if each element possesses an edge looping around on itself. The relation below is reflexive since we have a loop (a, a) for each $a \in A$ (Fig. 2.2).

Example 2.6 (Symmetric Relation) The graph of a symmetric relation will show for every arrow from a to b an opposite arrow from b to a. The relation in Fig. 2.3 is symmetric: i.e., whenever $(a, b) \in R$ then $(b, a) \in R$ (Fig. 2.3).

Example 2.7 (Transitive Relation) The graph of a transitive relation will show that whenever there is an arrow from a to b and an arrow from b to c that there is an arrow from a to c. The relation in Fig. 2.4 is transitive: i.e., whenever $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$ (Fig. 2.4).

Example 2.8 (Equivalence Relation) The relation on the set of integers $\mathbb{Z}$ defined by $(a, b) \in R$ if $a - b = 2k$ for some $k \in \mathbb{Z}$ is an equvalence relation, and partitions the set integers into two equivalence classes: i.e., the even and odd integers.

Domain and Range of Relation The domain of a relation R (A, B) is given by $\{a \in A | \exists b \in B$ and $(a, b) \in R\}$. It is denoted by **dom** R. The domain of the relation $R = \{(a, p), (a, r), (b, q)\}$ is $\{a, b\}$.

Fig. 2.4 Transitive relation
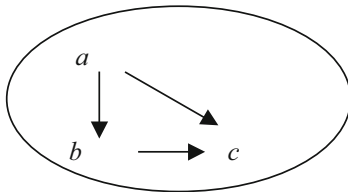
Fig. 2.5 Partitions of A
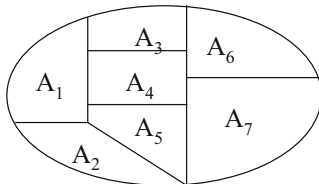
The range of a relation R (A, B) is given by $\{b \in B | \exists a \in A \text{ and } (a, b) \in R\}$. It is denoted by **rng** R. The range of the relation $R = \{(a, p), (a, r), (b, q)\}$ is $\{p, q, r\}$.

Inverse of a Relation Suppose $R \subseteq A \times B$ is a relation between A and B then the inverse relation $R^{-1} \subseteq B \times A$ is defined as the relation between B and A and is given by:

$$b R^{-1} a \text{ if and only if } a R b.$$

That is,

$$R^{-1} = \{(b, a) \in B \times A : (a, b) \in R\}.$$

Example 2.9 Let R be the relation between $\mathbb{Z}$ and $\mathbb{Z}^+$ defined by mRn if and only if $m^2 = n$. Then $R = \{(m, n) \in \mathbb{Z} \times \mathbb{Z}^+ : m^2 = n\}$ and $R^{-1} = \{(n, m) \in \mathbb{Z}^+ \times \mathbb{Z} : m^2 = n\}$.
For example, -3 R 9, -4 R 16, 0 R 0, 16 R^{-1} 4, 9 R^{-1} 3, etc.

Partitions and Equivalence Relations An equivalence relation on A leads to a partition of A, and vice versa for every partition of A there is a corresponding equivalence relation.

Let A be a finite set and let $A_1, A_2, \ldots, A_n$ be subsets of A such $A_i \neq \emptyset$ for all i, $A_i \cap A_j = \emptyset$ if $i \neq j$ and $A = \cup_i^n$ $A_i = A_1 \cup A_2 \cup \cdots \cup A_n$. The sets A_i partition the set A and these sets are called the classes of the partition (Fig. 2.5).

Theorem 2.2 An equivalence relation on A gives rise to a partition of A where the equivalence classes are given by $\text{Class}(a) = \{x | x \in A \text{ and } (a, x) \in R\}$. Similarly, a partition gives rise to an equivalence relation R, where $(a, b) \in R$ if and only if a and b are in the same partition.

Proof Clearly, $a \in \text{Class}(a)$ since R is reflexive and clearly the union of the equivalence classes is A. Next, we show that two equivalence classes are either equal or disjoint.

Suppose Class(a) ∩ Class(b) ≠ Ø. Let $x \in$ Class(a) ∩ Class(b) and so (a, x) and $(b, x) \in R$. By the symmetric property $(x, b) \in R$ and since R is transitive from (a, x) and (x, b) in R we deduce that $(a, b) \in R$. Therefore $b \in$ Class(a). Suppose y is an arbitrary member of Class (b) then $(b, y) \in R$ therefore from (a, b) and (b, y) in R we deduce that (a, y) is in R. Therefore since y was an arbitrary member of Class(a) we deduce that Class(b) ⊆ Class(a). Similarly, Class(a) ⊆ Class(b) and so Class(a) = Class(b).

This proves the first part of the theorem and for the second part we define a relation R such that $(a, b) \in R$ if a and b are in the same partition. It is clear that this is an equivalence relation.

2.3.2   Composition of Relations

The composition of two relations $R_1(A, B)$ and $R_2(B, C)$ is given by $R_2 \circ R_1$ where $(a, c) \in R_2 \circ R_1$ if and only there exists $b \in B$ such that $(a, b) \in R_1$ and $(b, c) \in R_2$. The composition of relations is associative: i.e.,

$$(R_3 \circ R_2) \circ R_1 = R_3 \circ (R_2 \circ R_1).$$

Example 2.10 (Composition of Relations) Consider a library that maintains two files. The first file maintains the serial number s of each book with details of the author a of the book. This may be represented by the relation $R_1 = s R_1 a$. The second file maintains the library card number c of its borrowers and the serial number s of any books that they have borrowed. This may be represented by the relation $R_2 = c R_2 s$.

The library wishes to issue a reminder to its borrowers of the authors of all books currently on loan to them. This may be determined by the composition of $R_1 \circ R_2$: i.e., $c R_1 \circ R_2 a$ if there is book with serial number s such that $s R_1 a$ and $c R_2 s$.

Example 2.11 (Composition of Relations) Consider sets $A = \{a, b, c\}$, $B = \{d, e, f\}$, $C = \{g, h, i\}$ and relations $R(A, B) = \{(a, d), (a, f), (b, d), (c, e)\}$ and $S(B, C) = \{(d, h), (d, i), (e, g), (e, h)\}$. Then we graph these relations and show how to determine the composition pictorially (Fig. 2.6).

$S \circ R$ is determined by choosing $x \in A$ and $y \in C$ and checking if there is a route from x to y in the graph. If so, we join x to y in $S \circ R$. For example, if we consider a and h we see that there is a path from a to d and from d to h and therefore (a, h) is in the composition of S and R.

$S \circ R$
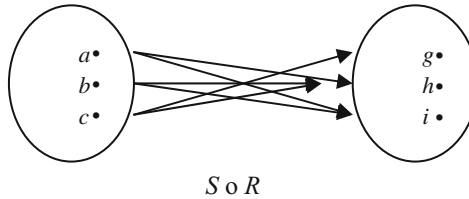
Fig. 2.6 Composition of relations

R(A,B)   S(B,C)
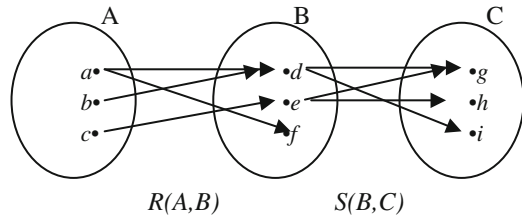
The union of two relations $R_1(A, B)$ and $R_2(A, B)$ is meaningful (as these are both subsets of $A \times B$). The union $R_1 \cup R_2$ is defined as $(a, b) \in R_1 \cup R_2$ if and only if $(a, b) \in R_1$ or $(a, b) \in R_2$.

Similarly, the intersection of R_1 and R_2 ($R_1 \cap R_2$) is meaningful and is defined as $(a, b) \in R_1 \cap R_2$ if and only if $(a, b) \in R_1$ and $(a, b) \in R_2$. The relation R_1 is a subset of $R_2(R_1 \subseteq R_2)$ if whenever $(a, b) \in R_1$ then $(a, b) \in R_2$.

The inverse of the relation R was discussed earlier and is given by the relation R^{-1} where $R^{-1} = \{(b, a)|(a, b) \in R\}$.

The composition of R and R^{-1} yields: $R^{-1} \circ R = \{(a, a)|a \in \text{dom } R\} = i_A$ and $R \circ R^{-1} = \{(b, b)|b \in \text{dom } R^{-1}\} = i_B$.

2.3.3 Binary Relations

A binary relation R on A is a relation between A and A, and a binary relation can always be composed with itself. Its inverse is a binary relation on the same set. The following are all relations on A:

$$R^2 = R \circ R,$$

$$R^3 = (R \circ R) \circ R,$$

$$R^0 = i_A \text{ (identity relation)},$$

$$R^{-2} = R^{-1} \circ R^{-1}.$$

Example 2.12 Let R be the binary relation on the set of all people P such that $(a, b) \in R$ if a is a parent of b. Then the relation R^n is interpreted as:

R is the parent relationship.
R^2 is the grandparent relationship.
R^3 is the great grandparent relationship.
R^{-1} is the child relationship.
R^{-2} is the grandchild relationship.
R^{-3} is the great grandchild relationship.

This can be generalized to a relation R^n on A where $R^n = R \circ R \circ \cdots \circ R$ (n-times). The transitive closure of the relation R on A is given by:

$$R^* = \cup_{i=0}^{\infty} R^i = R^0 \cup R^1 \cup R^2 \cup \ldots R^n \cup \ldots,$$

where R^0 is the reflexive relation containing only each element in the domain of R: i.e., $R^0 = i_A = \{(a, a)|a \in \mathbf{dom}\ R\}$.

The positive transitive closure is similar to the transitive closure except that it does not contain R^0. It is given by:

$$R^+ = \cup^\infty_{i=1} R^i = R^1 \cup R^2 \cup \ldots R^n \cup \ldots .$$

$a\ R^+\ b$ if and only if $a\ Rn\ b$ for some $n > 0$: *i.e.*, there exists $c_1, c_2, \ldots c_n \in A$ such that:

$$a R c_1, c_1 R c_2, \ldots, c_n R b.$$

Parnas[6] introduced the concept of the limited domain relation (LD-relation), and a LD relation L consists of an ordered pair (R_L, C_L) where R_L is a relation and C_L is a subset of Dom R_L. The relation R_L is on a set U and C_L is termed the competence set of the LD relation L. A description of LD relations and a discussion of their properties are in Chap. 2 of [Par:01].

The importance of LD relations is that they may be used to describe program execution. The relation component of the LD relation L describes a set of states such that if execution starts in state x it may terminate in state y. The set U is the set of states. The competence set of L is such that if execution starts in a state that is in the competence set then it is guaranteed to terminate.

2.4   Functions

A function $f : A \to B$ is a special relation such that for each element $a \in A$ there is exactly (or at most)[7] one element $b \in B$. This is written as $f(a) = b$.
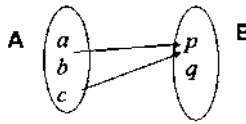
A function is a relation but not every relation is a function. For example, the relation in the diagram below is not a funcion since there are two arrows from the element $a \in A$.

---

[6] Parnas made important contributions to software engineering in the 1970s. He invented information hiding which is used in object-oriented design.

[7] We distinguish between total and partial functions. A total function $f:A \to B$ is defined for all elements in A whereas a partial function may be undefined for one or more elements in A.

Fig. 2.7 Domain and range of
a partial function

The domain of the function (denoted by **dom** f) is the set of values in A for which
the function is defined. The domain of the function is A provided that f is a total
function. The co-domain of the function is B. The range of the function (denoted **rng**
f) is a subset of the co-domain and consists of:

$$\mathbf{rng}\ f = \{r | r \in B \text{ such that } f(a) = r \text{ for some } a \in A\}.$$

Functions may be partial or total. A *partial function* (or partial mapping) may be
undefined for some values of A, and patial functions arise regularly in the computing
field. *Total functions* are defined for every value in A and many functions encountered
in mathematics are total (Fig. 2.7).

Example 2.13 (Functions) Functions are an essential part of mathematics and com-
puter science, and there are many well-known functions such as the trigonometric
functions $\sin(x)$, $\cos(x)$ and $\tan(x)$; the logarithmic function $\ln(x)$; the exponential
functions e^x and polynomial functions.

(i)   Consider the partial function $f : \mathbb{R} \rightarrow \mathbb{R}$ where

$$f(x) = \frac{1}{x} \quad \text{(where } x \neq 0\text{)}.$$

This partial function is defined everywhere except for $x = 0$.

(ii) Consider the function $f : \mathbb{R} \rightarrow \mathbb{R}$ where

$$f(x) = x^2.$$

Then this function is defined for all $x \in \mathbb{R}$.

   Partial functions often arise in computing as a program may be undefined or fail
to terminate for several values of its arguments (e.g., infinite loops). Care is required
to ensure that the partial function is defined for the argument to which it is to be
applied.

Consider a program P that has one natural number as its input and which for some input values will never terminate. Suppose that if it terminates it prints a single real result and halts. Then P can be regarded as a partial mapping from $\mathbb{N}$ to $\mathbb{R}$,

$$P : \mathbb{N} \to \mathbb{R}.$$

Example 2.14 How many total functions $f : A \to B$ are there from A to B (where A and B are finite sets)?

Each element of A maps to any element of B, i.e. there are $|B|$ choices for each element $a \in A$. Since there are $|A|$ elements in A the number of total functions is given by:

$$|B||B|\ldots|B| \qquad\qquad (|A|\text{times})$$

$$= |B|^{|A|} \qquad\qquad \text{total functions between } A \text{ and } B.$$

Example 2.15 How many partial functions $f : A \to B$ are there from A to B (where A and B are finite sets)?

Each element of A may map to any element of B or to no element of B (as it may not appear). In other words, there are $|B| + 1$ choices for each element of A. Since there are $|A|$ elements in A the number of distinct partial functions between A and B is given by:

$$(|B| + |1|)(|B| + |1|)\ldots(|B| + |1|) \qquad\qquad (|A|\text{times})$$
$$= (|B| + |1|)^{|A|}.$$

Two partial functions f and g are equal if:

1. dom $f =$ dom g.
2. $f(a) = g(a)$ for all $a \in$ dom f.

A function f is less defined than a function $g(f \subseteq g)$ if the domain of f is a subset of the domain of g, and the functions agree for every value on the domain of f:

1. dom $f \subseteq$ dom g.
2. $f(a) = g(a)$ for all $a \in$ dom f.

The composition of functions is similar to the composition of relations. Suppose $f : A \to \mathrm{B}$ and $g : B \to C$ then $g \circ f : A \to C$ is a function, and this is written as $g \circ f(x)$ or $g(f(x))$ for $x \in A$.

The composition of functions is not commutative and this can be seen by an example. Consider the function $f : R \to R$ such that $f(x) = x^2$ and the function $g : R \to R$ such that $g(x) = x + 2$. Then

$$g \circ f(x) = g(x^2) = x^2 + 2,$$

$$f \circ g(x) = f(x + 2) = (x + 2)^2 = x^2 + 4x + 4.$$

Clearly, $g \circ f(x) \neq f \circ g(x)$ and so composition of functions is not commutative. The composition of functions is associative, as the composition of relations is associative and every function is a relation. For $f : A \to B$, $g : B \to C$, and $h : C \to D$ we have:

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Fig. 2.8 Injective and
surjective functions

1-1, Not Onto   Onto, Not 1-1
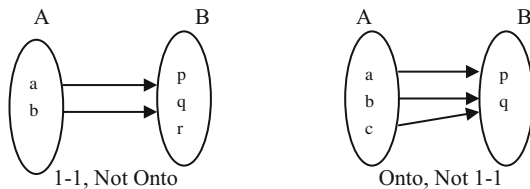
Fig. 2.9 Bijective function.
(One to one and onto)
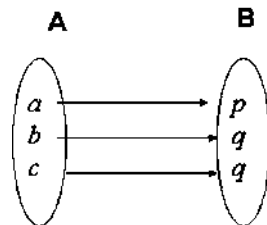
A function $f : A \rightarrow B$ is *injective* (*one to one*) if:

$$f(a_1) = f(a_2) \Rightarrow a_1 = a_2.$$

For example, consider the function $f : \mathbb{R} \rightarrow \mathbb{R}$ with $f(x) = x^2$. Then $f(3) = f(-3) = 9$ and so this function is not one to one.

A function $f : A \rightarrow B$ is *surjective* (*onto*) if given any $b \in B$ there exists an $a \in A$ such that $f(a) = b$. Consider the function $f : \mathbb{R} \rightarrow \mathbb{R}$ with $f(x) = x + 1$. Clearly, given any $r \in \mathbb{R}$ then $f(r - 1) = r$ and so f is onto (Fig. 2.8).

A function is *bijective* if it is one to one and onto. That is, there is a one to one correspondence between the elements in A and B, and for each $b \in B$ there is a unique $a \in A$ such that $f(a) = b$ (Fig. 2.9).

The inverse of a relation was discussed earlier and the relational inverse of a function $f : A \rightarrow B$ clearly exists. The relational inverse of the function may or may not be a function.

However, if the relational inverse is a function it is denoted by $f^{-1} : B \rightarrow A$. A total function has an inverse if and only if it is bijective whereas a partial function has an inverse if and only if it is injective.

The identity function $1_A : A \rightarrow A$ is a function such that $1_A(a) = a$ for all $a \in A$. Clearly, when the inverse of the function exists then we have that $f^{-1} \circ f = 1_A$ and $f^- \circ f^{-1} = 1_B$.

Theorem 2.3 A total function has an inverse if and only if it is bijective.

Proof Suppose $f : A \rightarrow B$ has an inverse f^{-1}. Then we show that f is bijective.
We first show that f is one to one. Suppose $f(x_1) = f(x_2)$ then

$$f^{-1}(f(x_1)) = f^{-1}(f(x_2)),$$
$$\Rightarrow f^{-1} \circ f(x_1) = f^{-1} \circ f(x_2),$$

$$\Rightarrow 1_A(x_1) = 1_A(x_2),$$
$$\Rightarrow x_1 = x_2.$$

Next we first show that f is onto. Let $b \in B$ and let $a = f^{-1}(b)$ then

$$f(a) = f(f^{-1}(b)) = b \text{ and so } f \text{ is surjective.}$$

The second part of the proof is concerned with showing that if $f : A \rightarrow B$ is bijective then it has an inverse f^{-1}. Clearly, since f is bijective we have that for each $a \in A$ there exists a unique $b \in B$ such that $f(a) = b$.

Define $g: B \rightarrow A$ by letting $g(b)$ be the unique a in A such that $f(a) = b$. Then we have that:

$$gof(a) = g(b) = a \text{ and } fog(b) = f(a) = b.$$

Therefore, g is the inverse of f.

2.5   Review Questions

1. What is a set? A relation? A function?
2. Explain the difference between a partial and a total function.
3. Explain the difference between a relation and a function.
4. Determine $A \times B$ where $A = \{a, b, c, d\}$ and $B = \{1, 2, 3\}$.
5. Determine the symmetric difference $A \triangle B$ where $A = \{a, b, c, d\}$ and $B = \{c, d, e\}$.
6. What is the graph of the relation $\leq$ on the set $A = \{2, 3, 4\}$.
7. What is the composition of S and R (i.e., $S \circ R$), where R is a relation between A and B, and S is a relation between B and C. The sets A, B, C are defined as $A = \{a, b, c, d\}$, $B = \{e, f, g\}$, $C = \{h, i, j, k\}$ and $R = \{(a, e), (b, e), (b, g), (c, e), (d, f)\}$ with $S = \{(e, h), (e, k), (f, j), (f, k), (g, h)\}$.
8. What is the domain and range of the relation R where $R = \{(a, p), (a, r), (b, q)\}$.
9. Determine the inverse relation R^{-1} where $R = \{(a, 2), (a, 5), (b, 3), (b, 4), (c, 1)\}$.
10. Determine the inverse of the function $f: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x) = \frac{x-2}{x-3}(x \neq 3)$ and $f(3) = 1$.
11. Give examples of injective, surjective and bijective functions.
12. Let $n \geq 2$ be a fixed integer. Consider the relation $\equiv$ defined by $\{(p, q) : p, q \in \mathbb{Z}, n|(q - p)\}$
    a. Show $\equiv$ is an equivalence relation.
    b. What are the equivalence classes of this relation?

2.6   Summary

This chapter provided an introduction to set theory, relations and functions. Sets are collections of well-defined objects, a relation between A and B indicates relationships between members of the sets A and B and functions are a special type of relation where there is at most one relationship for each element a in A with an element in B.

A set is a collection of well-defined objects that contains no duplicates. There are many examples of sets such as the set of natural numbers $\mathbb{N}$, the integer numbers $\mathbb{Z}$ and so on.

The Cartesian product allows a new set to be created from existing sets. The Cartesian product of two sets S and T (denoted $S \times T$) is the set of ordered pairs $\{(s, t) | s \in S, t \in T\}$.

A binary relation $R(A, B)$ is a subset of the Cartesian product $(A \times B)$ of A and B where A and B are sets. The domain of the relation is A and the co-domain of the relation is B. The notation aRb signifies that there is a relation between a and b and that $(a, b) \in R$. An n-ary relation $R(A_1, A_2, \ldots, A_n)$ is a subset of $(A_1 \times A_2 \times \cdots \times A_n)$.

A total function $f : A \to B$ is a special relation such that for each element $a \in A$ there is exactly one element $b \in B$. This is written as $f(a) = b$. A function is a relation but not every relation is a function.
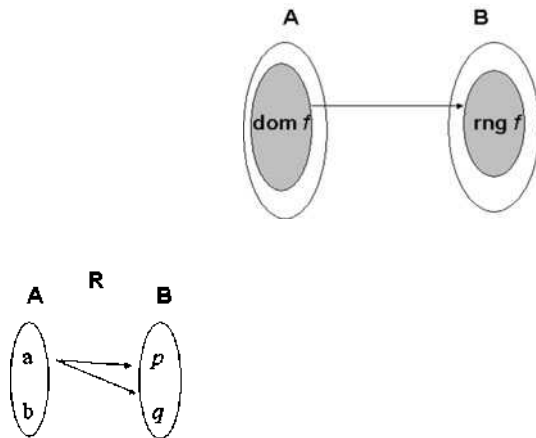
The domain of the function (denoted by $dom\ f$) is the set of values in A for which the function is defined. The domain of the function is A provided that f is a total function. The co-domain of the function is B.

Chapter 3
Logic

Key Topics

Syllogistic Logic
Fallacies
Propositional Logic
Truth Tables
Mathematical Proof
Natural Deduction
Predicate Logic
Universal and Existential Quantifiers
Temporal Logic
Undefined Values
Theorem Provers

3.1 Introduction

Logic is concerned with reasoning and with establishing the validity of arguments. It allows conclusions to be deduced from premises according to logical rules, and a valid deductive argument establishes the truth of the conclusion provided that the premises are true. Logic plays a key role in reasoning and deduction in mathematics but is regarded as a separate discipline from mathematics. There were attempts in the early twentieth century to show that all mathematics can be derived from formal logic. However, the Austrian logician, Kurt Goedel showed that there are truths in the formal system of arithmetic that cannot be proved within the formal system (i.e. first-order arithmetic is incomplete).

Early work on logic was done by Aristotle in the fourth century BC in the *Organon* [Ack:94]. Aristotle regarded logic as a useful tool of enquiry into any subject, and he developed *syllogistic logic*. This is a form of reasoning in which a conclusion is drawn from two premises, where each premise is in a subject–predicate form. A

Table 3.1 Types of syllogistic premises

Type	Symbol	Example
Universal affirmative	G *A* M	All Greeks are mortal
Universal negative	G *E* M	No Greek is mortal
Particular affirmative	G *I* M	Some Greek is mortal
Particular negative	G *O* M	Some Greek is not mortal

Table 3.2 Forms of syllogistic premises

	Form (i)	Form (ii)	Form (iii)	Form (iv)
Premise 1	M P	P M	P M	M P
Premise 2	M S	S M	M S	S M
Conclusion	S P	S P	S P	S P

common or middle term is present in the two premises but not in the conclusion. For example:

All Greeks are mortal
Socrates is a Greek
. . . .
Therefore Socrates is mortal

The common (or middle) term in this example is 'Greek'. It occurs in both premises but not in the conclusion. The above argument is valid and Aristotle studied and classified the various types of syllogistic arguments to determine those that were valid or invalid. Each premise contains a subject and a predicate, and the middle term may act as the subject or the predicate. Each premise is an affirmation or negative affirmation, and an affirmation may be universal or particular. The universal and particular affirmations and negatives are described in Table 3.1.

This leads to four basic forms of syllogistic arguments where the middle is the subject of both premises; the predicate of both premises; and the subject of one premise and the predicate of the other premise (Table 3.2).

There are four types of premises (A, E, I and O) and therefore 16 sets of premise pairs for each of the forms above. However, only some of these premise pairs will yield a valid conclusion. Aristotle went through every possible premise pair to determine if a valid argument may be derived. The syllogistic argument above is of form (iv) and is valid:

GAM
SIG
. . . . . .
SIM

Syllogistic logic is a "term-logic" with letters used to stand for the individual terms. Syllogistic logic was the first attempt at a science of logic and it remained in use up to the nineteenth century. There are many limitations to what it may express, and on its suitability as a representation of how the mind works.

Aristotle's also studied and classified bad arguments (known as *fallacies*) (Table 3.3). These include:

Table 3.3 Fallacies in arguments

Argument	Description/example
Hasty/accident generalisation	This is a bad argument that involves a generalisation that disregards exceptions.
Slippery slope	This argument outlines a chain reaction leading to a highly undesirable situation that will occur if a certain situation is allowed. The claim is that even if one step is taken onto the slippery slope, then we will fall all the way down to the bottom.
Against the person (*Ad Hominem*)	The focus of this argument is to attack the person rather than the argument that the person has made.
Appeal to people (*Ad Populum*)	This argument involves an appeal to popular belief to support an argument, with a claim that the majority of the population supports this argument. However, popular opinion is not always correct.
Appeal to authority (*Ad Verecundiam*)	This argument is when an appeal is made to an authoritative figure to support an argument, and where the authority is not an expert in this area.
Appeal to pity (*Ad Misericordiam*)	This is where the arguer tries to get people to accept a conclusion by making them feel sorry for someone.
Appeal to ignorance	The arguer makes the case that there is no conclusive evidence on the issue at hand and that therefore his conclusion should be accepted.
Straw man argument	The arguer sets up a version of an opponent's position of his argument and defeats this watered down version of his opponent's position.
Begging the question	This is a circular argument where the arguer relies on a premise that says the same thing as the conclusion and without providing any real evidence for the conclusion.
Red herring	The arguer goes off on a tangent that has nothing to do with the argument in question.
False dichotomy	The arguer presents the case that there are only two possible outcomes (often there are more). One of the possible outcomes is then eliminated leading to the desired outcome. The argument suggests that there is only one outcome.

Chrysippus who was the head of the Stoics in the third century BC developed an early version of propositional logic. He distinguished between simple and compound propositions and introduced a set of logical connectives. He also studied various logical argument forms such as *modus ponens* and *modus tollens*. His propositional logic did not replace Aristotle's syllogistic logic, and George Boole developed modern propositional logic in the nineteenth century. This is discussed in the next section.

3.2 Propositional Logic

Propositional logic is the study of propositions where a proposition is a statement that is either true or false. There are many examples of propositions such as "$1 + 1 = 2$" which is a true proposition, and the statement that 'Today is Wednesday' which is true if today is Wednesday and false otherwise. The statement $x > 0$ is not a proposition as it contains a variable x, and it is meaningful to consider its truth or falsity only

Table 3.4 Truth table for
formula W

A	B	$W(A, B)$
T	T	T
T	F	T
F	T	F
F	F	F

when a value is assigned to x. Once the variable x is assigned a value it becomes a proposition. The statement "This sentence is false" is not a proposition as if it is true it is false and is false it is true.

A propositional variable may be used to stand for a proposition (e.g. let the variable P stand for the proposition '$2 + 2 = 4$' which is a true proposition). A propositional variable takes the value true or false. The negation of a proposition P (denoted $\neg P$) is the proposition that is true if and only if P is false, and is false if and only if P is true.

A well-formed formula (WFF) in propositional calculus is a syntactically correct formula created according to the syntactic rules of the underlying calculus. A WFF is built up from variables, constants, terms, and logical connectives such as conjunction, disjunction, implication, equivalence and negation. A distinguished subset of these well-formed formulae are the axioms of the calculus, and there are rules of inference that allow the truth of new formulae to be derived from the axioms and from formulae that have already demonstrated to be true in the calculus.

A formula in propositional calculus may contain several propositional variables, and the truth or falsity of the individual variables needs to be known prior to determining the truth or falsity of the logical formula.

Each propositional variable has two possible values, and a formula with n-propositional variables has 2^n values associated with the propositional variables. The set of values associated with the n variables may be used to derive a truth table with 2^n rows and $n + 1$ columns. Each row gives each of the 2^n values that the n variables may take and column $n + 1$ gives the result of the logical expression for that set of values of the propositional variables. For example, the propositional formula W defined in the truth table above has two propositional variables A and B, with $2^2 = 4$ rows for each of the values that the two propositional variables may take. There are $2 + 1 = 3$ columns with W defined in the third column (Table 3.4).

A rich set of connectives is employed in the calculus to combine propositions and to build up the well-formed formulae. This includes the conjunction of two propositions $(A \wedge B)$; the disjunction of two propositions $(A \vee B)$; and the implication of two propositions $(A \Rightarrow B)$. These connectives allow compound propositions to be formed, and the truth of the compound propositions is determined from the truth values of its constituent propositions and the rules associated with the logical connective. The meaning of the logical connectives is given by truth tables.[1]

---

[1] Basic truth tables were first used by Frege, and developed further by Post and Wittgenstein.

Fig. 3.1 George Boole
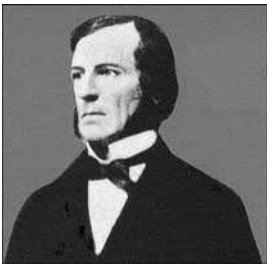
Table 3.5 Conjunction

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

Mathematical logic is concerned with the science of inference, and it involves proceeding in a methodical way from the axioms and using the rules of inference to derive further truths.

The rules of inference allow new propositions to be deduced from a set of existing propositions. A valid argument (or deduction) is truth preserving: i.e. for a valid logical argument if the set of premises is true, then the conclusion (i.e. the deduced proposition) will also be true. The rules of inference include rules such as *modus ponens*, and this rule states that given the truths of the proposition A and the proposition $A \Rightarrow B$, then the truth of proposition B may be deduced.

The propositional calculus is employed in reasoning about propositions and may be applied to formalise arguments in natural language. It has also being applied to computer science and the term 'Boolean algebra' is named after the George Boole who was a self-taught mathematician from Lincoln in England (Fig. 3.1). Boole was the first professor of mathematics at Queens College, Cork[2] in the mid-nineteenth century, and he formalised the laws of propositional logic that are the foundation for modern computers [ORg:12].

3.2.1 Truth Tables

Truth tables enable the truth value of a compound proposition to be determined from its underlying propositions. The truth of a compound proposition is determined from the truth of its constituent parts, and so the truth of a compound formula containing several propositional variables is determined from the underlying propositional variables and the logical connectives.

The conjunction of A and B (denoted $A \wedge B$) is true if and only if both A and B are true, and is false in all other cases (Table 3.5). The disjunction of two propositions A and B (denoted $A \vee B$) is true if at least one of A and B are true, and false in all other

---

[2] This institution is now known as University College Cork and has approximately 18,000 students.

Table 3.6 Disjunction

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

Table 3.7 Implication

A	B	$A \Rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

Table 3.8 Equivalence

A	B	$A \Leftrightarrow B$
T	T	T
T	F	F
F	T	F
F	F	T

Table 3.9 Not operation

A	$\neg A$
T	F
F	T

cases (Table 3.6). The disjunction operator is known as the 'inclusive or' operator as it is also true when both A and B are true; there is also an *exclusive or* operator that is true exactly when one of A or B is true, and is false otherwise.

Example 3.1 Consider proposition A given by "An orange is a fruit" and the proposition B given by "$2 + 2 = 5$", then A is true and B is false. Therefore,

1. $A \wedge B$ (i.e. An orange is a fruit and $2 + 2 = 5$) is false
2. $A \vee B$ (i.e. An orange is a fruit or $2 + 2 = 5$) is true

The implication operation ($A \Rightarrow B$) is true if whenever A is true means that B is also true, and also whenever A is false (Table 3.7). It is equivalent (as shown by a truth table) to $\neg A \vee B$. The equivalence operation ($A \Leftrightarrow B$) is true whenever both A and B are true, or whenever both A and B are false; it is false otherwise (Table 3.8).

The not operator ($\neg$) is a unary operator (i.e. it has one argument) and is such that $\neg A$ is true when A is false, and is false when A is true (Table 3.9).

Example 3.2 Consider proposition A given by "Jaffa cakes are biscuits" and the proposition B given by "$2 + 2 = 5$", then A is true and B is false. Therefore,

1. $A \Rightarrow B$ (i.e. Jaffa cakes are biscuits implies $2 + 2 = 5$) is false
2. $A \Leftrightarrow B$ (i.e. Jaffa cakes are biscuits is equivalent to $2 + 2 = 5$) is false
3. $\neg B$ (i.e. $2 + 2 \neq 5$) is true.

Creating a Truth Table The truth table for a WFF $W(P_1, P_2, \ldots, P_n)$ is a table with 2^n rows and $n + 1$ columns. Each row lists a different combination of truth values of

Table 3.10 Truth table for $W(P, Q, R)$

P	Q	R	$W(P, Q, R)$
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

the proposition variables $P_1, P_2, \ldots, P_n$ followed by the corresponding truth value of W (Table 3.10).

The example above gives the truth table for a formula W with three propositional variables (meaning that there are $2^3 = 8$ rows in the truth table).

3.2.2 Properties of Propositional Calculus

The propositional calculus has several nice properties such as the commutative, associative and distributive properties. These ease the evaluation of complex expressions and allow logical expressions to be simplified.

The *commutative property* holds for the conjunction and disjunction binary operators, and it states that the order of evaluation of the two propositions may be reversed without affecting the resulting truth value: i.e.

$$A \wedge B = B \wedge A$$
$$A \vee B = B \vee A$$

The *associative property* holds for the conjunction and disjunction operators. This means that order of evaluation of a sub-expression does not affect the resulting truth value: i.e.

$$(A \wedge B) \wedge C = A \wedge (B \wedge C)$$
$$(A \vee B) \vee C = A \vee (B \vee C)$$

The conjunction operator *distributes* over the disjunction operator and vice versa.

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$
$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

The result of the logical conjunction of two propositions is false if one of the propositions is false (irrespective of the value of the other proposition).

$$A \wedge F = F \wedge A = F$$

The result of the logical disjunction of two propositions is true if one of the propositions is true (irrespective of the value of the other proposition).

$$A \vee T = T \vee A = T$$

Table 3.11 Tautology $B \vee \neg B$

B	$\neg B$	$B \vee \neg B$
T	F	T
F	T	T

The result of the logical disjunction of two propositions, where one of the propositions is known to be false is given by the truth value of the other proposition. That is, the Boolean value 'F' acts as the identity for the disjunction operation.

$$A \vee F = A = F \vee A$$

The result of the logical conjunction of two propositions, where one of the propositions is known to be true, is given by the truth value of the other proposition. That is, the Boolean value 'T' acts as the identity for the conjunction operation.

$$A \wedge T = A = T \wedge A$$

The $\wedge$ and $\vee$ operators are *idempotent*. That is, when the arguments of the conjunction or disjunction operator are the same proposition A the result is A. The idempotent property allows expressions to be simplified.

$$A \wedge A = A$$
$$A \vee A = A$$

The *law of the excluded middle* is a fundamental property of the propositional calculus. It states that a proposition A is either true or false: i.e. there is no third logical value.

$$A \vee \neg A$$

De-Morgan was a contemporary of Boole in the nineteenth century, and the following law is known as De Morgan's law:

$$\neg(A \wedge B) = \neg A \vee \neg B$$
$$\neg(A \vee B) = \neg A \wedge \neg B$$

Certain well-formed formulae are true for all values of their constituent variables. This can be seen from the truth table when the last column of the truth table consists entirely of true values. A proposition that is true for all values of its constituent propositional variables is known as a *tautology*. An example of a tautology is the proposition $A \vee \neg A$ (Table 3.11).

A proposition that is false for all values of its constituent propositional variables is known as a *contradiction*. An example of a contradiction is the proposition $A \wedge \neg A$.

3.2.3   Proof in Propositional Calculus

Logic enables further truths to be derived from existing truths by rules of inference that are truth preserving. The propositional calculus is both *complete* and *consistent*.

The completeness property means that all true propositions are deducible in the calculus, and the consistency property means that there is no formula A such that both A and $\neg A$ are deducible in the calculus.

An argument in propositional logic consists of a sequence of formulae that are the premises of the argument and a further formula that is the conclusion of the argument. One elementary way to see if the argument is valid is to produce a truth table to determine if the conclusion is true whenever all of the premises are true.

Consider a set of premises $P_1, P_2, \ldots, P_n$ and conclusion Q. Then to determine if the argument is valid using a truth table involves adding a column in the truth table for each premise $P_1, P_2, \ldots, P_n$, and then identify the rows in the truth table for which these premises are all true. The truth value of the conclusion Q is examined in each of these rows, and if Q is true for each case for which $P_1, P_2, \ldots, P_n$ are all true, then the argument is valid. This is equivalent to $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q$ is a tautology.

An alternate approach to proof with truth tables is to assume the negation of the desired conclusion (i.e. $\neg Q$) and to show that the premises and the negation of the conclusion result in a contradiction (i.e. $P_1 \wedge P_2 \wedge \ldots \wedge P_n \wedge \neg Q$) is a contradiction. The use of truth tables becomes cumbersome when there are a large number of variables involved, as there are 2^n truth table entries for n propositional variables.

Procedure for Proof by Truth Table

1. Consider argument $P_1, P_2, \ldots, P_n$ with conclusion Q.
2. Draw truth table with column in truth table for each premise $P_1, P_2, \ldots, P_n$.
3. Identify rows in truth table for which these premises are all true.
4. Examine the truth value of Q for these rows.
5. If Q is true for each case that $P_1, P_2, \ldots, P_n$ are true, then the argument is valid.
6. That is $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q$ is a tautology.

Truth tables provide an informal approach to proof and the proof is provided in terms of the meanings of the propositions and logical connectives. The formalisation of propositional logic includes the definition of an alphabet of symbols and well-formed formulae of the calculus, the axioms of the calculus and rules of inference for logical deduction.

The deduction of a new formulae Q is via a sequence of well-formed formulae $P_1, P_2, \ldots, P_n$ (where $P_n = Q$) such that each P_i is either an axiom, a hypothesis or deducible from an earlier pair of formulae P_j, P_k, (where P_k is of the form $P_j \Rightarrow P_i$) and modus ponens. Modus ponens is a rule of inference that states that given propositions A, and $A \Rightarrow B$, then proposition B may be deduced. The deduction of a formula Q from a set of hypothesis H is denoted by $H \vdash Q$ and where Q is deducible from the axioms alone this is denoted by $\vdash Q$.

The deduction theorem states that if $H \cup \{P\} \vdash Q$, then $H \vdash P \Rightarrow Q$ and the converse of the theorem is also true: i.e. if $H \vdash P \Rightarrow Q$, then $H \cup \{P\} \vdash Q$. The axiomatic approach (due to the German mathematician, David Hilbert) allows reasoning about symbols according to rules, and to derive theorems from formulae irrespective of the meanings of the symbols and formulae.

Table 3.12 Logical equivalence of two WFFs

P	Q	$P \wedge Q$	$\neg P$	$\neg Q$	$\neg P \vee \neg Q$	$\neg(P \neg \vee \neg Q)$
T	T	T	F	F	F	T
T	F	F	F	T	T	F
F	T	F	T	F	T	F
F	F	F	T	T	T	F

Table 3.13 Logical implication of two WFFs

$P\,Q\,R$	$(P \wedge Q) \vee$ $(Q \wedge \neg R)$	$Q \vee R$
T T T	T	T
T T F	T	T
T F T	F	T
T F F	F	F
F T T	F	T
F T F	T	T
F F T	F	T
F F F	F	F

Propositional calculus is *sound*; i.e. any theorem derived using the Hilbert approach is true. Further, the calculus is also complete and every tautology has a proof (i.e. is a theorem in the formal system). The propositional calculus is consistent: (i.e. it is not possible that both the WFF A and $\neg A$ are deducible in the calculus).

Propositional calculus is *decidable*: i.e. there is an algorithm to determine for any WFF A, whether A is a theorem of the formal system. The Hilbert style system is slightly cumbersome in conducting proof and is quite different from the normal use of logic in mathematical deduction.

Logical Equivalence and Logical Implication The laws of mathematical reasoning are truth preserving, and are concerned with deriving further truths from existing truths. Logical reasoning is concerned with moving from one line in mathematical argument to another, and involves deducing the truth of another statement Q from the truth of P.

The statement Q may be in some sense be logically equivalent to P and this allows the truth of Q to be immediately deduced. In other cases, the truth of P is sufficiently strong to deduce the truth of Q; in other words, P logically implies Q. This leads naturally to a discussion of the concepts of logical equivalence ($W_1 \equiv W_2$) and logical implication ($W_1 \vdash W_2$).

Logical Equivalence Two well-formed formulae W_1 and W_2 with the same propositional variables ($P, Q, R \ldots$) are logically equivalent ($W_1 \equiv W_2$), if they are always simultaneously true or false for any given truth values of the propositional variables (Table 3.12).

If two well-formed formulae are logically equivalent, then it does not matter which of W_1 and W_2 is used, and $W_1 \Leftrightarrow W_2$ is a tautology. In the above-mentioned example, we see that $P \wedge Q$ is logically equivalent to $\neg(\neg P \vee \neg Q)$.

Logical Implication For two well-formed formulae W_1 and W_2 with the same propositional variables ($P, Q, R \ldots$) W_1 logically implies W_2 ($W_1 \vdash W_2$) if any assignment to the propositional variables which makes W_1 true also makes W_2 true.

Example 3.3 Show by truth tables that $(P \wedge Q) \vee (Q \wedge \neg R) \vdash (Q \vee R)$ (Table 3.13).

Table 3.14 Natural deduction rules

Rule	Definition	Description
$\wedge\,$I	$\dfrac{P_1, P_2, \ldots, P_n}{P_1 \wedge P_2 \wedge \ldots \wedge P_n}$	Given the truth of propositions $P_1, P_2, \ldots, P_n$, then the truth of the conjunction $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ follows. This rule shows how conjunction can be introduced
$\wedge\,$E	$\dfrac{P_1 \wedge P_2 \wedge \ldots \wedge P_n}{P_i}$ where $i \in \{1, \ldots, n\}$	Given the truth the conjunction $P_1 \wedge P_2 \wedge \ldots \wedge P_n$, then the truth of proposition P_i follows. This rule shows how a conjunction can be eliminated
$\vee\,$I	$\dfrac{P_i}{P_1 \vee P_2 \vee \ldots \vee P_n}$	Given the truth of propositions P_i, then the truth of the disjunction $P_1 \vee P_2 \vee \ldots \vee P_n$ follows. This rule shows how a disjunction can be introduced
$\vee\,$E	$\dfrac{P_1 \vee \ldots \vee P_n, P_1 \Rightarrow E, \ldots P_n \Rightarrow E}{E}$	Given the truth of the disjunction $P_1 \vee P_2 \vee \ldots \vee P_n$, and that each disjunct implies E, then the truth of E follows. This rule shows how a disjunction can be eliminated
$\Rightarrow\,$I	$\dfrac{\text{From } P_1, P_2, \ldots, P_n \text{ infer } P}{(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \Rightarrow P}$	This rule states that if we have a theorem that allows P to be inferred from the truth of premises $P_1, P_2, \ldots, P_n$ (or previously proved), then we can deduce $(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \Rightarrow P$. This is known as the *Deduction Theorem*
$\Rightarrow\,$E	$\dfrac{P_i \Rightarrow P_j, P_i}{P_j}$	This rule is known as *Modus Ponens*. The consequence of an implication follows if the antecedent is true (or has been previously proved)
$\equiv\,$I	$\dfrac{P_i \Rightarrow P_j, P_j \Rightarrow P_i}{P_i \equiv P_j}$	If proposition P_i implies proposition P_j and vice versa, then they are equivalent (i.e. $P_i \equiv P_j$)
$\equiv\,$E	$\dfrac{P_i \equiv P_j}{P_i \Rightarrow P_j, P_j \Rightarrow P_i}$	If proposition P_i is equivalent to proposition P_j, then proposition P_i implies proposition P_j and vice versa
$\neg\,$I	$\dfrac{\text{From } P \text{ infer } P_1 \wedge \neg P_1}{\neg P}$	If the proposition P allows a contradiction to be derived, then $\neg P$ is deduced. This is an example of a proof by contradiction
$\neg\,$E	$\dfrac{\text{From } \neg P \text{ infer } P_1 \wedge \neg P_1}{P}$	If the proposition $\neg P$ allows a contradiction to be derived, then P is deduced. This is an example of a proof by contradiction

The formula $(P \wedge Q) \vee (Q \wedge \neg R)$ is true on rows 1, 2 and 6 and formula $(Q \vee R)$ is also true on these rows. Therefore, $(P \wedge Q) \vee (Q \wedge \neg R) \vdash (Q \vee R)$. In other words, logical implication allows the truth of another statement W_2 to be deduced from the truth of W_1. Logical implication and logical equivalence allow the move from one line of a mathematical argument to another.

The German mathematician, Gerhard Gentzen, developed a method for logical deduction known as 'Natural Deduction', and this approach is closer to natural reasoning. It includes rules for $\wedge$, $\vee$, $\Rightarrow$ introduction and elimination and also for *reductio ab adsurdum*. There are ten inference rules in the Natural Deduction system and they include two inference rules for each of the five logical operators $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\equiv$.

The two inference rules per operator are the introduction rule (I) and the elimination rule (E). The rules are defined in Table 3.14.

Natural deduction may be employed in logical reasoning and is described in detail in [Gri:81, Kel:97].

3.2.4   Applications of Propositional Calculus

Propositional calculus may be employed in reasoning with arguments in natural language. First, the premises and conclusion of the argument are identified and formalised into propositions. Propositional logic is then employed to determine if the conclusion is a valid deduction from the premises. Consider, for example, the following argument that aims to prove that Superman does not exist.

> If Superman were able and willing to prevent evil, he would do so. If Superman were unable to prevent evil he would be impotent; if he were unwilling to prevent evil he would be malevolent; Superman does not prevent evil. If superman exists he is neither malevolent nor impotent; therefore Superman does not exist.

First, letters are employed to represent the propositions as follows:

a: Superman is able to prevent evil
w: Superman is willing to prevent evil
i: Superman is impotent
m: Superman is malevolent
p: Superman prevents evil
e: Superman exists

Then, the argument above is formalised in propositional logic as follows:

P_1 $(a \wedge w) \Rightarrow p$
P_2 $(\neg a \Rightarrow i) \wedge (\neg w \Rightarrow m)$
P_3 $\neg p$
P_4 $e \Rightarrow \neg i \wedge \neg m$
. . . .
Conclusion $P_1 \wedge P_2 \wedge P_3 \wedge P_4 \Rightarrow \neg e$

Proof that Superman Does not Exist

1.	$a \wedge w \Rightarrow p$	*Premise* 1
2.	$(\neg a \Rightarrow i) \wedge (\neg w \Rightarrow m)$	*Premise* 2
3.	$\neg p$	*Premise* 3
4.	$e \Rightarrow (\neg i \wedge \neg m)$	*Premise* 4
5.	$\neg p \Rightarrow \neg(a \wedge w)$	1, *Contrapositive*
6.	$\neg(a \wedge w)$	3, 5 *Modus ponens*
7.	$\neg a \vee \neg w$	6, *De Morgan's Law*
8.	$\neg(\neg i \wedge \neg m) \Rightarrow \neg e$	4, *Contrapositive*
9.	$i \vee m \Rightarrow \neg e$	8, *De Morgan's Law*
10.	$\neg a \Rightarrow i$	2, $\wedge$ *Elimination*
11.	$\neg w \Rightarrow m$	2, $\wedge$ *Elimination*
12.	$\neg\neg a \vee i$	10, $A \Rightarrow B$ equivalent to $\neg A \vee B$
13.	$\neg\neg a \vee i \vee m$	11, $\vee$ Introduction
14.	$\neg\neg a \vee (i \vee m)$	

15.	$\neg a \Rightarrow (i \vee m)$	14, $A \Rightarrow B$ equivalent to $\neg A \vee B$
16.	$\neg\neg w \vee m$	11, $A \Rightarrow B$ equivalent to $\neg A \vee B$
17.	$\neg\neg w \vee (i \vee m)$	
18.	$\neg w \Rightarrow (i \vee m)$	17, $A \Rightarrow B$ equivalent to $\neg A \vee B$
19.	$(i \vee m)$	7, 15, 18 $\vee Elimination$
20.	$\neg e$	9, 19 *Modus ponens*

Second Proof

1.	$\neg p$	P_3
2.	$\neg(a \wedge w) \vee p$	P_1 ($A \Rightarrow B \equiv \neg A \vee B$)
3.	$\neg(a \wedge w)$	1, 2 $A \vee B, \neg B \vdash A$
4.	$\neg a \vee \neg w$	3, De Morgan's Law
5.	$(\neg a \Rightarrow i)$	P_2 ($\wedge$-Elimination)
6.	$\neg a \Rightarrow i \vee m$	5, $x \Rightarrow y \vdash x \Rightarrow y \vee z$
7.	$\neg w \Rightarrow m$	P_2 ($\wedge$-Elimination)
8.	$\neg w \Rightarrow i \vee m$	7, $x \Rightarrow y \vdash x \Rightarrow y \vee z$
9.	$(\neg a \vee \neg w) \Rightarrow (i \vee m)$	8, $x \Rightarrow z, y \Rightarrow z \vdash x \vee y \Rightarrow z$
10.	$i \vee m$	4, 9 Modus Ponens
11.	$e \Rightarrow \neg(i \vee m)$	P_4 (De Morgan's Law)
12.	$\neg e \vee \neg(i \vee m)$	11, ($A \Rightarrow B \equiv \neg A \vee B$)
13.	$\neg e$	10, 12 $A \vee B, \neg B \vdash A$

Therefore, the conclusion that Superman does not exist is a valid deduction from the given premises.

3.2.5   Limitations of Propositional Calculus

The propositional calculus deals with propositions only. It is incapable of dealing with the syllogism "All Greeks are mortal; Socrates is a Greek; therefore Socrates is mortal" discussed earlier. This would be expressed in propositional calculus as three propositions A, B therefore C, where A stands for 'All Greeks are mortal', B stands for 'Socrates is a Greek' and C stands for 'Socrates is mortal'. Propositional logic does not allow the conclusion that all Greeks are mortal to be derived from the other two premises.

Predicate calculus deals with these limitations by employing variables and terms, and using universal and existential quantification to express that a particular property is true of all (or at least one) values of a variable. Predicate calculus is discussed in the next section.

3.3   Predicate Calculus

Predicate logic allows complex facts about the world to be represented, and new facts about the world may be derived in a way that guarantees that if the initial facts are true, then the conclusions are true. Predicate calculus includes predicates, variables and quantifiers.

A *predicate* is a characteristic or property that the subject of a statement can have, and it may include variables. These statements with variables become propositions once the variables are assigned values. The set of values that the variables may take is termed the 'universe of discourse', and the variables take values from this set.

Predicate calculus employs quantifiers to express properties such as all members of the domain have a particular property: e.g. $(\forall x)Px$, or that there is at least one member that has a particular property: e.g. $(\exists x)Px$. These are referred to as the *universal and existential quantifiers.*

The syllogism 'All Greeks are mortal; Socrates is a Greek; therefore Socrates is mortal' may be easily expressed in predicate calculus by:

$(\forall x)(Gx \Rightarrow Mx)$
Gs
. . . .
Ms

In this example, the predicate Gx stands for x is a Greek and the predicate Mx stands for x is mortal. The formula $Gx \Rightarrow Mx$ states that if x is a Greek, then x is mortal. The formula $(\forall x)(Gx \Rightarrow Mx)$ states for any x, if x is a Greek, then x is mortal. The formula Gs states that Socrates is a Greek and the formula Ms states that Socrates is mortal.

Example 3.4 (Predicates) A predicate may have one or more variables. A predicate that has only one variable (a 1-place predicate) is often related to sets; a predicate with two variables (a 2-place predicate) is often related to relations and a predicate with n variables (an n-place predicate) is a n-ary relation. Propositions do not contain variables and so they are 0-place predicates. The following are examples of predicates:

1. The predicate Prime(x) states that x is a prime number (with the natural numbers being the universe of discourse).
2. *Lawyer(a)* may stand for a is a lawyer.
3. Mean(m,x,y) states that m is the mean of x and y: i.e. $m = 1/2(x+y)$.
4. $x < 6$ states that x is less than 6.
5. G(x, y) states that x is greater than y.
6. LE(x, y) states that x is less than or equal to y.
7. Real(x) states that x is a real number.
8. Father(x, y) states that x is the father of y.

The predicate calculus is built from an alphabet of constants, variables, function letters, predicate letters and logical connectives (including the logical connectives discussed in propositional logic and universal and existential quantifiers). Terms are built from constants, variables and function letters. A constant or variable is a term, and if $t_1, t_2, \ldots, t_k$ are terms, then $f_i^k(t_1, t_2, \ldots, t_k)$ is a term (where f_i^k is a k-ary function letter). Examples of terms include:

π where π is the constant 3.14159.
x^2 where x is a variable and square is a 1-ary function letter.

$x^2 + y^2$ where $x^2 + y^2$ is shorthand for the function add(square(x), square(y)) where
add is a 2-ary function letter and square is a 1-ary function letter.

The well-formed formulae are built from terms as follows. If P_i^k is a k-ary predicate
letter, $t_1, t_2, \ldots, t_k$ are terms, then $P_i^k (t_1, t_2, \ldots, t_k)$ is a WFF. If A and B are well-
formed formulae, then so are $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $A \equiv B$, $(\forall x)A$ and $(\exists x)A$.
Examples of well-formed formulae include:

$$x = y$$

$$(\forall x)(x > 2)$$

$$(\exists x)x^2 = 2$$

$$(\forall x)(x > 2 \wedge x < 10)$$

$$(\exists y)x^2 = y.$$

Universal and Existential Quantification   The universal quantifier is used to express
a statement such as that all members of the domain have property P. This is written as
$(\forall x) P(x)$ and expresses the statement that the property $P(x)$ is true for all x. Similarly,
$(\forall x_1, x_2, \ldots, x_n) P(x_1, x_2, \ldots, x_n)$ states that property $P(x_1, x_2, \ldots, x_n)$ is true for
all $x_1, x_2, \ldots, x_n$. Clearly, the predicate $(\forall x) P(a, b)$ is identical to $P(a, b)$ since it
contains no variables, and the predicate $(\forall y \in N) (x \leq y)$ is true if $x = 1$ and false
otherwise.

The existential quantifier states that there is at least one member in the domain
of discourse that has property P. This is written as $(\exists x)P(x)$, and the predicate $(\exists x_1,
x_2, \ldots, x_n) P(x_1, x_2, \ldots, x_n)$ states that there is at least one value of $(x_1, x_2, \ldots, x_n)$
such that $P(x_1, x_2, \ldots, x_n)$.

Example 3.5 (Quantifiers)

1. $(\exists p)$ (Prime$(p) \wedge p > 1{,}000{,}000$) is true
   It expresses the fact that there is at least one prime number greater than a million,
   and this is clearly true since there are an infinite number of primes.
2. $(\forall x)(\exists y) \, x < y$ is true
   This predicate expresses the fact that given any number x we can always find a
   larger number: e.g. $y = x + 1$.
3. $(\exists y)(\forall x) \, x < y$ is false
   This predicate expresses the statement that there is a natural number y such that
   all natural numbers are less than y. Clearly, this statement is false since there is
   no largest natural number and so the predicate $(\exists y) (\forall x) \, x < y$ is false.

Comment 3.1 *There is a need to be careful with the order in which quantifiers are
written as the meaning of a statement may be completely changed by the simple
transposition of two quantifiers.*

The formula $x = y$ states that x is the same as y; $(\forall x)(x > 2)$ states that every value
of x is greater than the constant 2; $(\exists x) \, x^2 = 2$ states that there is an x such that the
value of x is the square root of 2 and $(\forall x)(\exists y) \, x^2 = y$ states that for every x there is a
y such that the square of x is y.

The definition of terms and well-formed formulae specifies the syntax of the predicate calculus, and the set of well-formed formulae gives the language of the predicate calculus. The terms and well-formed formulae are built from the symbols and these symbols are not given meaning in the formal definition of the syntax. The language defined by the calculus needs to be given an interpretation in order to give a meaning to the terms and formulae of the calculus. The interpretation needs to define the domain of values of the constants and variables, provide meaning to the function letters, the predicate letters and the logical connectives. The formalisation of predicate calculus is discussed in the next section.

3.3.1   Formalisation of Predicate Calculus

The formalisation of predicate calculus includes the definition of an alphabet of symbols (including constants and variables), the definition of function and predicate letters, logical connectives and quantifiers. This leads to the definitions of the terms and well-formed formulae of the calculus.

There is a set of axioms for predicate calculus and two rules of inference used for the deduction of new formulae from the existing axioms and previously deduced formulae. The deduction of a new formula Q is via a sequence of well-formed formulae $P_1, P_2, \ldots, P_n$ (where $P_n = Q$) such that each P_i is either an axiom, a hypothesis or deducible from one or more of the earlier formulae in the sequence.

The two rules of inference are *modus ponens* and *generalisation*. Modus ponens is a rule of inference that states that given predicate formulae A, and $A \Rightarrow B$, then the predicate formula B may be deduced. Generalisation is a rule of inference that states that given predicate formula A, then the formula $(\forall x) A$ may be deduced where x is any variable.

The deduction of a formula Q from a set of hypothesis H is denoted by $H \vdash Q$ and where Q is deducible from the axioms alone this is denoted by $\vdash Q$. The deduction theorem states that if $H \cup \{P\} \vdash Q$, then $H \vdash P \Rightarrow Q$[3] and the converse of the theorem is also true: i.e. if $H \vdash P \Rightarrow Q$, then $H \cup \{P\} \vdash Q$.

The approach allows reasoning about symbols according to rules, and to derive theorems from formulae irrespective of the meanings of the symbols and formulae. However, the predicate calculus is sound: i.e. any theorem derived is true, and the calculus is also complete.

Scope of Quantifiers The scope of the quantifier $(\forall x)$ in the WFF $(\forall x)A$ is A. Similarly, the scope of the quantifier $(\exists x)$ in the WFF $(\exists x)B$ is B. The variable x that occurs within the scope of the quantifier is said to be a bound variable. If a variable is not within the scope of a quantifier it is *free*.

---

[3] This is stated more formally that if $H \cup \vdash \{P\}\ Q$ by a deduction containing no application of generalization to a variable that occurs free in P then $H \vdash P \Rightarrow Q$.

Example 3.6 (Scope of Quantifiers)

1. x is free in the WFF $\forall y \, (x^2 + y > 5)$.
2. x is bound in the WFF $\forall x \, (x^2 > 2)$.

A WFF is *closed* if it has no free variables. The substitution of a term t for x in A can only take place when no free variable in t will become bound by a quantifier in A through the substitution. Otherwise, the interpretation of A would be altered by the substitution.

A term t is free for x in A if no free occurrence of x occurs within the scope of a quantifier $(\forall y)$ or $(\exists y)$ where y is free in t. This means that the term t may be substituted for x without altering the interpretation of the WFF A.

For example, suppose A is $\forall y \, (x^2 + y^2 > 2)$ and the term t is y, then t is not free for x in A as the substitution of t for x in A will cause the free variable y in t to become bound by the quantifier $\forall y$ in A.

3.3.2   Interpretation and Valuation Functions

An interpretation gives meaning to a formula and consists of a domain of discourse and a valuation function. If the formula is a sentence (i.e. does not contain any free variables), then the given interpretation of the formula is either true or false. If a formula has free variables, then the truth or falsity of the formula depends on the values given to the free variables. A formula with free variables essentially describes a relation say, $R(x_1, x_2, \ldots, x_n)$ such that $R(x_1, x_2, \ldots, x_n)$ is true if $(x_1, x_2, \ldots, x_n)$ is in relation R. If the formula is true irrespective of the values given to the free variables, then the formula is true in the interpretation.

A *valuation* (meaning) *function* gives meaning to the logical symbols and connectives. Thus, associated with each constant c is a constant c_Σ in some universe of values Σ; with each function symbol f of arity k, we have a function symbol f_Σ in Σ and $f_\Sigma \colon \Sigma^k \to \Sigma$; and for each predicate symbol P of arity k a relation $P_\Sigma \subseteq \Sigma^k$. *The valuation function, in effect, gives the semantics to the language of the predicate calculus L.* The truth of a predicate P is then defined in terms of the meanings of the terms, the meanings of the functions, predicate symbols and the normal meanings of the connectives.

Mendelson [Men:87] provides a technical definition of truth in terms of satisfaction (with respect to an interpretation M). Intuitively, a formula F is *satisfiable* if it is *true* (in the intuitive sense) for some assignment of the free variables in the formula F. If a formula F is satisfied for every possible assignment to the free variables in F, then it is *true* (in the technical sense) for the interpretation M. An analogous definition is provided for *false* in the interpretation M.

A formula is *valid* if it is true in every interpretation; however, as there may be an uncountable number of interpretations, it may not be possible to check this requirement in practice. M is said to be a model for a set of formulae if and only if every formula is true in M.

There is a distinction between proof-theoretic and model-theoretic approaches in predicate calculus. *Proof theoretic* is essentially syntactic, and there is a list of axioms with rules of inference. The theorems of the calculus are logically derived (i.e. $\vdash A$) and the logical truths are as a result of the syntax or form of the formulae, rather than the *meaning* of the formulae. In contrast, *model theoretic* is essentially semantic. The truth derives from the meaning of the symbols and connectives, rather than the logical structure of the formulae. This is written as $\vdash_M A$.

A calculus is *sound* if all of the logically valid theorems are true in the interpretation, i.e. proof theoretic $\Rightarrow$ model theoretic. A calculus is complete if all the truths in an interpretation are provable in the calculus, i.e. model theoretic $\Rightarrow$ proof theoretic. A calculus is *consistent* if there is no formula A such that $\vdash A$ and $\vdash \neg A$.

The predicate calculus is sound, complete and consistent. *Predicate calculus is not decidable*: i.e. there is no algorithm to determine for any WFF A whether A is a theorem of the formal system. The undecidability of the predicate calculus may be demonstrated by showing that if the predicate calculus is decidable, then the halting problem (of Turing machines) is solvable.

3.3.3 Properties of Predicate Calculus

The followings are properties of the predicate calculus:

1. $(\forall x)P(x) \equiv (\forall y)P(y)$.
2. $(\forall x)P(x) \equiv \neg(\exists x)\neg P(x)$.
3. $(\exists x)P(x) \equiv \neg(\forall x)\neg P(x)$.
4. $(\exists x)P(x) \equiv (\exists y)P(y)$.
5. $(\forall x)(\forall y)P(x, y) \equiv (\forall y)(\forall x)P(x, y)$.
6. $(\exists x)P(x) \vee Q(x) \equiv (\exists x)P(x) \vee (\exists y)Q(y)$.
7. $(\forall x)P(x) \wedge Q(x) \equiv (\forall x)P(x) \wedge (\forall y)Q(y)$.

3.3.4 Applications of Predicate Calculus

The predicate calculus may be employed to formally state the system requirements of a proposed system. It may be used to conduct formal proof to verify the presence or absence of certain properties in a specification. It may also be employed to define piecewise defined functions such as $f(x, y)$ where $f(x, y)$ is defined by:

$$f(x, y) = x^2 - y^2 \qquad \text{where } x \leq 0 \wedge y < 0;$$
$$f(x, y) = x^2 + y^2 \qquad \text{where } x > 0 \wedge y < 0;$$
$$f(x, y) = x + y \qquad \text{where } x \geq 0 \wedge y = 0;$$
$$f(x, y) = x - y \qquad \text{where } x < 0 \wedge y = 0;$$
$$f(x, y) = x + y \qquad \text{where } x \leq 0 \wedge y > 0;$$
$$f(x, y) = x^2 + y^2 \qquad \text{where } x > 0 \wedge y > 0.$$

The predicate calculus may be employed for *program verification* and to show that a code fragment fulfils its specification. The objective of program verification is to show that if the pre-condition is true before execution of the code fragment, then this implies that the post-condition is true after execution of the code fragment.

A program fragment *a* is *partially correct* for pre-condition *P* and post-condition *Q* if and only if whenever *a* is executed in any state in which *P* is satisfied and execution terminates, then the resulting state satisfies *Q*. Partial correctness is denoted by *P*{*F*}*Q*, and Hoare's Axiomatic Semantics is based on partial correctness. It requires proof that the post-condition is satisfied if the program terminates.

A program fragment *a* is *totally correct* for pre-condition *P* and post-condition *Q* if and only if whenever *a* is executed in any state in which *P* is satisfied, then the execution terminates and the resulting state satisfies *Q*. It is denoted by {*P*}*F*{*Q*}, and Dijkstra's calculus of weakest pre-conditions is based on total correctness. It is required to prove that if the pre-condition is satisfied, then the program terminates and the post-condition is satisfied.

3.4 Undefined Values

Total functions $f: X \rightarrow Y$ are functions that are defined for every element in their domain, and they are widely used in mathematics. However, there are functions that are undefined for one or more elements in their domain, and one example is the function $y = 1/x$. This function s undefined for $x = 0$.

Partial functions arise naturally in computer science, and such functions may fail to be defined for one or more values in their domain. One approach to dealing with partial functions is to employ a pre-condition, which limits the application of the function to the restricted members of the domain for which the function is defined. This makes it possible to define a new set (a proper subset of the domain of the function) for which the function is total over the new set.

Undefined terms often arise[4] and need to be dealt with. Consider, the example of the square root function $\sqrt{x}$ taken from [Par:93]. The domain of this function is the positive real numbers, and the following expression is undefined:

$$(x > 0) \wedge (y = \sqrt{x}) \vee (x \leq 0) \wedge (y = \sqrt{-x}).$$

The reason this is undefined is since the usual rules for evaluating such an expression involves evaluating each sub-expression, and then performing the Boolean operations. However, when $x < 0$ the sub-expression $y = \sqrt{x}$ is undefined, whereas when $x > 0$ the sub-expression $y = \sqrt{-x}$ is undefined. Clearly, it is desirable that such expressions be handled, and that for the example above, the expression would evaluate to true.

Classical two-valued logic does not handle this situation adequately and there have been several proposals to deal with undefined values. Dijkstra's approach is to

---

[4] It is best to avoid undefinedness by taking care with the definitions of terms and expressions.

Fig. 3.2 Conjunction

Q	T	F	⊥
P		**P∧Q**	
T	T	F	⊥
F	F	F	F
⊥	⊥	F	⊥

Fig. 3.3 Disjunction

Q	T	F	⊥
P		**P∨Q**	
T	T	T	T
F	T	F	⊥
⊥	T	⊥	⊥

Fig. 3.4 Implication

Q	T	F	⊥
P		**P⇒Q**	
T	T	F	⊥
F	T	T	T
⊥	T	⊥	⊥

use the *cand* and *cor* operators in which the value of the left hand operand determines whether the right hand operand expression is evaluated or not. Jone's logic of partial functions (LPFs) [Jon:86] uses a three-valued logic[5] and Parnas's[6] approach is an extension to the predicate calculus to deal with partial functions that preserves the two-valued logic.

3.4.1   Logic of Partial Functions

Jones [Jon:86] has proposed the logic of partial functions (LPFs) as an approach to deal with terms that may be undefined. This is a three-valued logic and a logical term may be true, false or undefined (denoted ⊥). The definition of the truth-functional operators used in classical two-valued logic is extended to three-valued logic. The truth table is defined below.

The conjunction of *P* and *Q* is true when both *P* and *Q* are true; false if one of *P* or *Q* is false and undefined otherwise (Fig. 3.2). The operation is commutative. The disjunction of *P* and *Q* ($P \lor Q$) is true if one of *P* or *Q* is true; false if both *P* and *Q* are false and undefined otherwise (Fig. 3.3). The implication operation ($P \Rightarrow Q$) is true when *P* is false or when *Q* is true; false when *P* is true and *Q* is false and undefined otherwise (Fig. 3.4). The equivalence operation ($P \equiv Q$) is true when both *P* and *Q* are true or false; it is false when *P* is true and *Q* is false (and vice versa) and it is undefined otherwise (Fig. 3.5).

---

[5] The above expression would evaluate to true under Jones three-valued logic of partial functions.

[6] The above expression evaluates to true for Parnas logic (a two-valued logic).

Fig. 3.5 Equivalence

Q	T	F	⊥
P		P≡Q	
T	T	F	⊥
F	F	T	⊥
⊥	⊥	⊥	⊥

Fig. 3.6 Negation

A	¬A
T	F
F	T
⊥	⊥

The not operator ($\neg$) is a unary operator such as $\neg A$ is true when A is false, false when A is true and undefined when A is undefined (Fig. 3.6).

The result of an operation may be known immediately after knowing the value of one of the operands (e.g. disjunction is true if P is true irrespective of the value of Q). The law of the excluded middle: i.e. $A \vee \neg A$ does not hold in the three-valued logic, and Jones [Jon:86] argues that this is reasonable as one would not expect the following to be true:

$$\left(\frac{1}{0} = 1\right) \vee \left(\frac{1}{0} \neq 1\right).$$

There are other well-known laws that fail to hold such as:

1. $E \Rightarrow E$.
2. Deduction theorem $E_1 \vdash E_2$ does not justify $\vdash E_1 \Rightarrow E_2$ unless it is known that E_1 is defined.
3. Many of the tautologies of standard logic.

3.4.2 Parnas Logic

Parnas's approach to logic is based on the classical two-valued logic, and his philosophy is that truth values should be true or false only,[7] and that there is no third logical value. It is an extension to predicate calculus to deal with partial functions. The evaluation of a logical expression yields the value 'true' or 'false' irrespective of the assignment of values to the variables in the expression. This allows the expression: $(y = \sqrt{x}) \vee (y = \sqrt{-x})$ that is undefined in classical logic to yield the value true.

The advantages of his approach are that no new symbols are introduced into the logic, and that the logical connectives retain their traditional meaning. This makes it easier for engineers and computer scientists to understand, as it is closer to their intuitive understanding of logic.

---

[7] It is a little strange to assign the value false to the primitive predicate calculus expression $y = \frac{1}{0}$.

Table 3.15 Examples of
Parnas evaluation of
undefinedness

Expression	$x < 0$	$x \geq 0$
$y = \sqrt{x}$	False	True if $y = \sqrt{x}$, false otherwise
$y = {}^1/_0$	False	False
$y = x^2 + \sqrt{x}$	False	True if $y = x^2 + \sqrt{x}$, false otherwise

Table 3.16 Example of
undefinedness in array

Expression	$i \in \{1 \ldots N\}$	$i \notin \{1 \ldots N\}$
$B[i] = x$	True if $B[i] = x$	False
$\exists i, B[i] = x$	True if $B[i] = x$ for some i, false otherwise	False

Fig. 3.7 Finding index in array

The meaning of predicate expressions is given by first defining the meaning of the primitive expressions. These are then used as the building blocks for predicate expressions. The evaluation of a primitive expression $R_j(V)$ (where V is a comma-separated set of terms with some elements of V involving the application of partial functions) is false if the value of an argument of a function used in one of the terms of V is not in the domain of that function.[8] The following examples (Table 3.15 and 3.16) should make this clearer.

These primitive expressions are used to build the predicate expressions, and the standard logical connectives are used to yield truth values for the predicate expression.

The power of Parnas logic may be seen by considering a tabular expressions example [Par:93]. The table below specifies the behaviour of a program that searches the array B (Fig. 3.7) for the value x. It describes the properties of the values of j' and *present'*. There are two cases to consider:

1. There is an element in the array with the value of x.
2. There is no such element in the array with the value of x.

It is clear from the above-mentioned example that the predicate expressions $\exists i, B[i] = x$ and $\neg (\exists i, B[i] = x)$ are defined. One disadvantage of the Parnas approach is that some common relational operators (e.g. $>$, $\geq$, $\leq$ and $<$) are not primitive in the logic. However, these relational operators are then constructed from primitive operators. Further, the axiom of reflection does not hold in the logic. Parnas logic is defined in detail in [Par:93].

---

[8] The approach avoids the undefined logical value ($\perp$) and preserves the two-valued logic.

Fig. 3.8 Edsger Dijkstra.
(Courtesty of Brian Randell)

Table 3.17 *a cand b*

A	B	a cand b
T	T	T
T	F	F
T	U	U
F	T	F
F	F	F
F	U	F
U	T	U
U	F	U
U	U	U

3.4.3 Dijkstra and Undefinedness

The *cand* and *cor* operators were introduced by Dijkstra (Fig. 3.8) to deal with undefined values. They are non-commutative operators and allow the evaluation of predicates that contain undefined values.

Consider the following expression:

$$y = 0 \vee (x/y = 2).$$

Then this expression is undefined when $y = 0$ as x/y is undefined, since the logical disjunction operation is not defined when one of its operands is undefined. However, there is a case for giving meaning to such an expression when $y = 0$, since in that case the first operand of the logical or operation is true. Further, the logical *disjunction* operation is defined to be true if either of its operands is true. This motivates the introduction of the *cand* and *cor* operators. These operators are associative and their truth tables (Table 3.17 and 3.18) are defined below:

The order of the evaluation of the operands for the *cand* operation is to *evaluate the first operand*; if the first operand is true, then the result of the operation is the second operand; otherwise the result is false. The expression *a cand b* is equivalent to:

$$a \ cand \ b \cong \text{if } a \text{ then } b \text{ else } F.$$

Table 3.18 *a cor b*

A	B	a cor b
T	T	T
T	F	T
T	U	T
F	T	T
F	F	F
F	U	U
U	T	U
U	F	U
U	U	U

The order of the evaluation of the operands for the *cor* operation is to evaluate the first operand. If the first operand is true, then the result of the operation is true; otherwise the result of the operation is the second operand. The expression *a cor b* is equivalent to:

$$a \; cor \; b \cong \text{if } a \text{ then } T \text{ else } b.$$

The *cand* and *cor* operators are not commutative but satisfy the following laws:

- *Associativity*
  The *cand* and *cor* operators are associative.

$$(A \; cand \; B) \; cand \; C = A \; cand \; (B \; cand \; C)$$
$$(A \; cor \; B) \; cor \; C = A \; cor \; (B \; cor \; C)$$

- *Distributivity*
  The *cand* operator distributes over the *cor* operator and vice versa.

$$A \; cand \; (B \; cor \; C) = (A \; cand \; B) \; cor \; (A \; cand \; C)$$
$$A \; cor \; (B \wedge C) = (A \; cor \; B) \; cand \; (A \; cor \; C)$$

De Morgan's law enables logical expressions to be simplified.

$$\neg(A \; cand \; B) = \neg A \; cor \; \neg B$$
$$\neg(A \; cor \; B) = \neg A \; cand \; \neg B$$

3.5   Other Logics

Temporal logic is concerned with the expression of properties that have time dependencies, and the various temporal logics can express facts about the past, present and future. It has been applied to specify temporal properties in natural language, artificial intelligence and in the specification and verification of program and system behaviour.

The roots of temporal logic lie in the Tense logic introduced by Prior in the 1960s and developed further by logicians and computer scientists. Tense logic contains four modal operators that express events in the future or in the past:

- *P* (It has at some time been the case that).
- *F* (It will be at some time be the case that).
- *H* (It has always been the case that).
- *G* (It will always be the case that).

P and *F* are known as *weak tense* operators with *H* and *G* known as *strong tense* operators. Temporal logics are applicable in the specification of computer systems, and a specification may require *safety*, *fairness* and *liveness properties* to be expressed. For example, a fairness property may state that it will always be the case that a certain property will hold sometime in the future. The specification of temporal properties often involves the use of special temporal operators.

Common temporal operators that are used are those to express properties that it will always be true; properties that will eventually be true; and a property that will be true in the next time instance. For example,

□ *P–P* is always true.
◇ *P–P* will be true sometime in the future.
○ *P–P* is true in the next time instant (*discrete time*).

It is also possible to express temporal operations directly in classical mathematics, and Parnas prefers this approach. He is critical of computer scientists for introducing unnecessary formalisms when classical mathematics already possesses the ability to do this. For example, the value of a function *f* at a time instance prior to the current time *t* is defined as:

$$\text{Prior}(f, t) = \lim_{\varepsilon \to 0} f(t - \varepsilon).$$

Another logic that arises in computer science is *fuzzy logic*, and this logic is used to deal with degrees of truth. The reader is referred to texts on temporal logic and fuzzy logics.

Perhaps, one of the more unusual logics that has been invented is *intuitionist logic* which was developed by Brouwer. This constructive approach to the foundations of mathematics was highly controversial as its acceptance as a foundation would have led to the rejection of many accepted theorems in classical mathematics. Brouwer was a Dutch mathematician who did important work in topology as well as in the foundations of mathematics, and a well-known fixpoint theorem in topology is named after him.

Brouwer was deeply interested in resolving the problems that arose from the paradoxes of set theory and in providing a solid foundation for mathematics. He took the extreme view that the proof of existence of a mathematical object must be constructive, and he rejected accepted mathematical practices such as the Law of the Excluded Middle and indirect proofs. He argued that for an entity to exist that it needed to be constructed.

Consequently, if the Brouwer view of the world were accepted, then many of the classical theorems of mathematics (including his own well-known results in topology) could no longer be said to be true. He developed a form of logic called Intuitionist Logic in which many of the results of classical mathematics were no longer true. The reader is referred to [Hey:66]. This logic has been applied to Type Theory by Martin Löf [Lof:84].

3.6   Tools for Logic

The formal verification of computer system often involves lengthy proofs consisting of several thousand formulae. Manual proof is error prone and there are several tools available to support theorem proving. These include the Boyer–Moore theorem prover known as NQTHM, the Isabelle theorem prover and the HOL system.

B.S. Boyer and J.S. Moore developed the Boyer–Moore theorem prover in the early 1970s. It has been improved since then and it is currently known as NQTHM (it has been superseded by ACL2 available from the University of Texas).

It has been effective in proving well-known theorems such as Goedel's Incompleteness Theorem, the insolvability of the Halting problem, a formalisation of the Motorola MC 68020 Microprocessor and many more.

Computational Logic Inc. was a company founded by Boyer and Moore in 1983 to share the benefits of a formal approach to software development with the wider computing community. It was based in Austin, Texas, and provided services in the mathematical modelling of hardware and software systems. This involved the use of mathematics and logic to formally specify microprocessors and other systems. The use of its theorem prover was to formally verify that the implementation meets its specification: i.e. to prove that the microprocessor or other system satisfies its specification.

Isabelle is a theorem-proving environment developed at Cambridge University by Larry Paulson and Tobias Nipkow of the Technical University of Munich. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas. The main application is the formalisation of mathematical proofs, and proving the correctness of computer hardware or software with respect to its specification, and proving properties of computer languages and protocols.

Isabelle is a generic theorem prover in the sense that it has the capacity to accept a variety of formal calculi, whereas most other theorem provers are specific to a specific formal calculus. It is available free of charge under an open source license.

The HOL system is an environment for interactive theorem proving in a higher order logic. The HOL system has been applied to the formalisation of mathematics and the verification of hardware. It was originally developed at Cambridge University in the United Kingdom in the early 1980s, and HOL 4 is the latest version and is an open source project. It is used by academia and industry.

There is a steep learning curve with the theorem provers above and it generally takes a couple of months for users to become familiar with them.

3.7 Review Questions

1. What is logic? A proposition? A predicate?
2. Give examples of fallacies in arguments in natural language (e.g. in politics, marketing and debates).
3. Draw a truth table to show that $\neg(P \Rightarrow Q) \equiv P \wedge \neg Q$.
4. Translate the sentence "Execution of program P begun with $x < 0$ will not terminate" into propositional form.
5. Prove the following theorems using the inference rules of natural deduction:
   a. From b infer $b \vee \neg c$.
   b. From $b \Rightarrow (c \wedge d)$, b infer d.
6. Explain the difference between the universal and the existential quantifier.
7. Express the following statements in the predicate calculus:
   a. All natural numbers are greater than 10.
   b. There is at least one natural number between 5 and 10.
   c. There is a prime number between 100 and 200.
8. Which of the following predicates are true?
   a. $\forall i \in \{10, \ldots, 50\}\ i^2 < 2000 \wedge i < 100$.
   b. $\exists i \in \mathbb{N}\ i > 5 \wedge i^2 = 25$.
   c. $\exists i \in \mathbb{N}\ i^2 = 25$.
9. Discuss the problem of undefinedness and the advantages and disadvantages of three-valued logics. Describe the approaches of Parnas, Dijkstra and Jones.
10. Show how the temporal operators may be expressed in classical mathematics. Discuss the merits of temporal operators.
11. Investigate the Isabelle (or another) theorem-proving environment and determine the extent to which it may assist with proof.

3.8 Summary

This chapter considered propositional and predicate calculus. Propositional logic is the study of propositions, and a proposition is a statement that is either true or false. A formula in propositional calculus may contain several variables, and the truth or falsity of the individual variables, and the meanings of the logical connectives determines the truth or falsity of the logical formula.

A rich set of connectives is employed in propositional calculus to combine propositions and to build up the well-formed formulae of the calculus. This includes the conjunction of two propositions $(A \wedge B)$, the disjunction of two propositions $(A \vee B)$ and the implication of two propositions $(A \Rightarrow B)$. These connectives allow compound propositions to be formed, and the truth of the compound propositions is determined from the truth values of the constituent propositions and the rules associated with the

logical connectives. The meaning of the logical connectives is given by truth tables. Predicates are statements involving variables and these statements become propositions once the variables are assigned values. Predicate calculus allows expressions such as all members of the domain have a particular property to be expressed formally: e.g. $(\forall x)Px$, or that there is at least one member that has a particular property: e.g. $(\exists x)Px$. Predicate calculus may be employed to specify the requirements for a proposed system and to give the definition of a piecewise-defined function.

The problem of undefinedness was discussed and solutions such as Jone's LPFs; Dijkstra's *cand* and *cor* operators and Parnas's extension to classical two-valued logic approach to undefinedness. Finally, temporal logic and its application to specifying properties with time dependencies were discussed.

Chapter 4
Software Engineering

> **Key Topics**
> Birth of Software Engineering
> Software Engineering Mathematics
> Floyd
> Hoare
> Formal Methods
> Software Inspections and Testing
> Project Management
> Software Process Maturity Models

4.1 Introduction

The NATO Science Committee organised two famous conferences on software engineering in the late 1960s. The first conference was held in Garmisch, Germany, in 1968, and it was followed by a second conference in Rome in 1969. The Garmisch conference was attended by more than 50 people from 11 countries.

The conferences highlighted the problems that existed in the software sector in the late 1960s, and the term "software crisis" was coined to refer to these problems. These included budget and schedule overruns of projects, and problems with the quality and reliability of the delivered software. This conference led to the birth of *software engineering* as a separate discipline, and the realisation that programming is quite distinct from science and mathematics. Programmers are like engineers in the sense that they design and build products; however, they need an appropriate education to design and develop software.[1]

---

[1] Software companies that are following approaches such as the CMM or ISO 9000:2000 consider the qualification of staff before assigning staff to performing specific tasks. The qualifications and experience required for the role are considered prior to appointing a person to carry out a particular role. Mature companies place significant emphasis on the education and continuous development of their staff, and in introducing best practice in software engineering into their organisation.

The construction of bridges was problematic in the nineteenth century, and many people who presented themselves as qualified to design and construct bridges did not have the required knowledge and expertise. Consequently, many bridges collapsed, endangering the lives of the public. This led to legislation requiring an engineer to be licensed by the Professional Engineering Association prior to practicing as an engineer. These engineering associations identify a core body of knowledge that the engineer is required to possess, and the licensing body verifies that the engineer has the required qualifications and experience. The licensing of engineers by most branches of engineering ensures that only personnel competent to design and build products actually do so. This in turn leads to products that the public can safely use. In other words, the engineer has a responsibility to ensure that the products are properly built and are safe for the public to use.

Parnas argues that traditional engineering be contrasted with the software engineering discipline where there is no licensing mechanism, and where individuals with no qualifications can participate in the design and building of software products.[2] However, companies today place a strong emphasis on qualifications and training.

The Standish Group has conducted research since the late 1990s [Std:99] on the extent of problems with schedule and budget overruns of IT projects. This study is conducted in the United States, but there is no reason to believe that European or Asian companies perform any better. The results indicate serious problems with on-time delivery, cost overruns and quality.[3] Fred Brooks has argued that software is inherently complex, and that there is no silver bullet that will resolve all of the problems associated with software projects such as schedule overruns and software quality problems [Brk:75, Brk:86].

Poor-quality software can at best cause minor irritation to clients, and in some circumstances it may seriously disrupt the work of the client organisation leading to injury or even the death of individuals (e.g. as in the case of the Therac-25[4] radiotherapy machine). The Y2K problem occurred due to poor design, as the representation of the date used two digits to record the year rather than four. Its correction required major rework, as it was necessary to examine all existing software code to determine how

---

There is a growing trend among companies to mature their software processes to enable them to deliver superior results. One of the purposes that the original CMM served was to enable the U.S. Department of Defence (DOD) to have a mechanism to assess the capability and maturity of software subcontractors.

[2] Modern HR recruitment specifies the requirements for a particular role, and interviews with candidates aim to establish that the candidate is suitably qualified, and has the appropriate experience for the role.

[3] It should be noted that these are IT projects covering diverse sectors including banking, telecommunications, etc., rather than pure software companies. Mature software companies using the CMM tend to be more consistent in project delivery with high quality.

[4] Therac-25 was a radiotherapy machine produced by the Atomic Energy of Canada Limited (AECL). It was involved in at least six accidents between 1985 and 1987 in which patients were given massive overdoses of radiation. The dose given was more than 100 times the intended dose and three of the patients died from radiation poisoning. These accidents highlighted the dangers of software control of safety-critical systems. The investigation subsequently highlighted the poor software design of the system and the poor software development practices employed.

the date was represented and to make appropriate corrections. Clearly, well-designed programs would have hidden the representation of the date thereby minimising the changes required for year 2000 compliance. The quality of software produced by mature software companies committed to continuous improvement tends to be superior.

Mathematics plays a key role in engineering and may potentially assist software engineers deliver high-quality software products that are safe to use. Several mathematical approaches that can assist in delivering high-quality software are described in [ORg:06]. There is a lot of industrial interest in software process maturity for software organisations, and approaches to assess and mature software companies are described in [ORg:02, ORg:10].[5] These focus mainly on improving the effectiveness of the management, engineering and organisation practices related to software engineering.

4.2   What is Software Engineering?

Software engineering involves multi-person construction of multi-version programs. The Institute of Electrical and Electronics Engineers (IEEE) 610.12 definition states that:

Definition 4.1 (Software Engineering)   *Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software, and the study of such approaches*.

Software engineering includes:

1. Methodologies to determine requirements, design, develop, implement and test software to meet customers' needs.
2. The philosophy of engineering: i.e. an engineering approach to developing software is adopted. That is, products are properly designed, developed, tested, with quality and safety properly addressed.
3. Mathematics[6] may be employed to assist with the design and verification of software products. The level of mathematics to be employed will depend on the safety-critical nature of the product, as systematic peer reviews and testing are often sufficient in building quality into the software product.
4. Sound project management and quality management practices are employed.

---

[5] Approaches such as the CMM or structured process improvement for construction environments (SPICE; ISO 15504) focus mainly on the management and organisational practices required in software engineering. The emphasis is on defining and following the software process. In practice, there is often insufficient technical detail on requirements, design, coding and testing in the models, as the models focus on what needs to be done rather how it should be done.

[6] There is no consensus at this time as to the appropriate role of mathematics in software engineering. My view is that the use of mathematics should be mandatory in the safety-critical and security-critical fields as it provides an extra level of quality assurance in these important fields.

Fig. 4.1 David Parnas

Software engineering requires the engineer to state precisely the requirements that the software product is to satisfy, and then to produce designs that will meet these requirements. Engineers provide a precise description of the problem to be solved; they then proceed to producing a design and validating its correctness; finally, the design is implemented and testing is performed to verify the correctness of the implementation with respect to the requirements. The software requirements needs to be unambiguous, and should clearly state what is and what is not required.

Classical engineers produce the product design, and then analyse their design for correctness. They use mathematics in their analysis, as this is the basis of confirming that the specifications are met. The level of mathematics employed will depend on the particular application and calculations involved. The term "engineer" is generally applied only to people who have attained the necessary education and competence to be called engineers, and who base their practice on mathematical and scientific principles. Often in computer science, the term "engineer" is employed rather loosely to refer to anyone who builds things, rather than to an individual with a core set of knowledge, experience and competence.

Parnas[7] is a strong advocate of the classical engineering approach, and he argues that computer scientists should have the right education to apply scientific and mathematical principles to their work. This includes mathematics and design, to enable them to be able to build high-quality and safe products. Baber has argued [Bab:11] that mathematics is the language of engineering. He argues that students should be shown how to turn a specification into a program using mathematics (Fig. 4.1).

Parnas has argued that computer science courses tend to include a small amount of mathematics, whereas mathematics is a significant part of an engineering course and is the language of classical engineering. He argues that students are generally taught programming languages and syntax, but not how to design and analyse software. He advocates a solid engineering approach to the teaching of mathematics with an emphasis on its application to developing and analysing product designs.

---

[7] Parnas has made key contributions to software engineering including information hiding, which is used in the object-oriented world. He has also done work (mainly of interest to the safety-critical field) on mathematical approaches to software quality.

He argues that software engineers need education on engineering mathematics; specification and design; converting designs into programs; software inspections, and testing. The education should enable the software engineer to produce well-designed programs that will correctly implement the requirements.

He argues that software engineers have individual responsibilities as professional engineers.[8] They are responsible for designing and implementing high-quality and reliable software that is safe to use. They are also accountable for their own decisions and actions[9] and have a responsibility to object to decisions that violate professional standards. Professional engineers have a duty to their clients to ensure that they are solving the real problem of the client. Engineers need to be honest about current capabilities, especially when asked to work on problems that have no appropriate technical solution. In another words, they should be honest and avoid accepting a contract for something that cannot be done.

The licensing of a professional engineer provides confidence that the engineer has the right education and experience to build safe and reliable products. Otherwise, the profession gets a bad name as a result of poor work carried out by unqualified people. Professional engineers are required to follow rules of good practice, and to object when the rules are violated.[10] The professional engineering body is responsible for enforcing standards and certification. The term "engineer" is a title that is awarded on merit, but it also places responsibilities on its holder.

Engineers have a professional responsibility and are required to behave ethically with their clients. The membership of the professional engineering body requires the member to adhere to the code of ethics of the profession. Most modern companies have a code of ethics that employees are required to adhere to. It details the required ethical behaviour and responsibilities.

---

[8] The concept of accountability is not new; indeed the ancient Babylonians employed a code of laws c. 1750 BC known as the Hammarabi Code. This code included the law that if a house collapsed and killed the owner then the builder of the house would be executed.

[9] However, it is unlikely that an individual programmer would be subject to litigation in the case of a flaw in a program causing damage or loss of life. A comprehensive disclaimer of responsibility for problems rather than a guarantee of quality accompany most software products. Software engineering is a team-based activity involving several engineers in various parts of the project, and it could be potentially difficult for an outside party to prove that the cause of a particular problem is due to the professional negligence of a particular software engineer, as there are many others involved in the process such as reviewers of documentation and code and the various test groups. Companies are more likely to be subject to litigation, as a company is legally responsible for the actions of their employees in the workplace, and the fact that a company is a financially richer entity than one of its employees. However, the legal aspects of licensing software may protect software companies from litigation including those companies that seem to place little emphasis on software quality. However, greater legal protection for the customer can be built into the contract between the supplier and the customer for bespoke-software development.

[10] Software companies that are following the CMMI or ISO 9000 will employ auditors to verify that the rules and best practice have been followed. Auditors report their findings to management and the findings are addressed appropriately by the project team and affected individuals.

Fig. 4.2 Waterfall lifecycle
model (V-model)
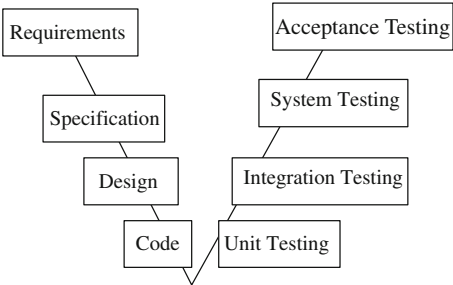
The approach used in current software engineering is to follow a well-defined software engineering process. The process includes activities such as project management, requirements gathering, requirements specification, architecture design, software design, coding and testing. Most companies use a set of templates for the various phases. The waterfall model [Roy:70] and spiral model [Boe:88] are popular software development lifecycles.

The waterfall model (Fig. 4.2) starts with requirements, followed by specification, design, implementation and testing. It is typically used for projects where the requirements can be identified early in the project lifecycle or are known in advance. The waterfall model is also called the "V" life cycle model, with the left-hand side of the "V" detailing requirements, specification, design and coding and the right-hand side detailing unit tests, integration tests, system tests and acceptance testing. Each phase has entry and exit criteria that must be satisfied before the next phase commences. There are several variations of the waterfall model.

The spiral model (Fig. 4.3) is useful where the requirements are not fully known at project initiation. There is an evolution of the requirements during the development which proceeds in a number of spirals, with each spiral typically involves updates to
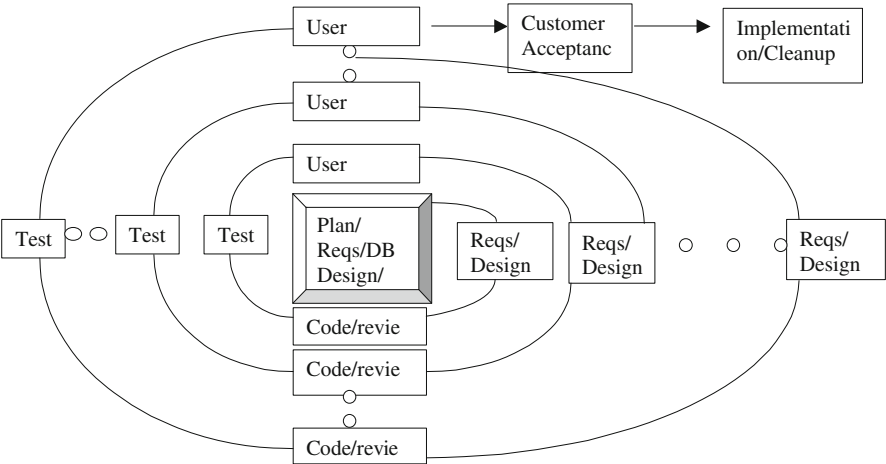
Fig. 4.3 Spiral lifecycle model
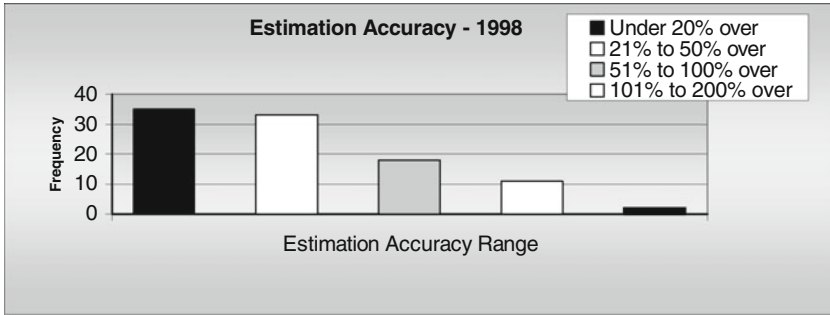
Fig. 4.4 Standish Group report: estimation accuracy

the requirements, design, code, testing, and a user review of the particular iteration or spiral.

The spiral is, in effect, a re-usable prototype and the customer examines the current iteration and provides feedback to the development team to be included in the next spiral. The approach is to partially implement the system. This leads to a better understanding of the requirements of the system and it then feeds into the next cycle in the spiral. The process repeats until the requirements and product are fully complete.

There are other lifecycle models: for example, the Cleanroom approach to software development includes a phase for formal specification and its approach to testing is quite distinct from other models, as it is based on the predicted usage of the software product. Finally, the Rational Unified Process (RUP) has become popular in recent years.

The challenge in software engineering is to deliver high-quality software on time to customers. The Standish Group research (Fig. 4.4) on project cost overruns in the United States during 1998 indicate that 33 % of projects are between 21 and 50 % overestimate, 18 % are between 51 and 100 % overestimate and 11 % of projects are between 101 and 200 % overestimate.

Accurate project estimation of cost and effort are the key challenges, and organisations need to determine how good their current estimation process actually is and to make improvements as appropriate. The use of software metrics allows effort estimation accuracy to be determined by computing the variance between actual project effort and the estimated project estimate.

Many companies today employ formal project management methodologies such as Prince 2 or Project Management Professional (PMP). These methodologies allow projects to be rigorously managed and include processes for initiating a project, planning a project, executing a project, monitoring and controlling a project and closing a project.

Risk management is a key part of project management, and its objective is to identify potential risks to the project; determine the probability of the risks occurring; assessing the impact of each risk if it materialises; identifying actions to eliminate the risk or to reduce its probability of occurrence; contingency plans in place to address

the risk if it materialises and finally, to track and manage the risks throughout the project.

The concept of process maturity has become popular with the Capability Maturity Model (CMM) developed by the Software Engineering Institute (SEI). The SEI has collected empirical data to suggest that there is a close relationship between software process maturity and the quality and the reliability of the delivered software. However, the main focus of the Capability Maturity Model Integration (CMMI) is on management and organisation practices rather than on technical engineering practices.

The implementation of the CMMI helps to provide a good engineering approach, as it places strict requirements on the characteristics of the underlying management and engineering processes that a company needs to have in place. The processes employed include:

- Developing and managing requirements.
- Design activities.
- Configuration Management.
- Selection and management of suppliers.
- Planning and managing projects.
- Building quality into the product with peer reviews.
- Performing rigorous testing.
- Performing independent audits.

There has been a growth of popularity among software developers in lightweight methodologies such as extreme programming (XP) [Bec:00]. These methodologies view documentation with distaste, and often software development commences prior to the full specification of the requirements.

4.3   Early Software Engineering

Robert Floyd was born in New York in 1936, and attended the University of Chicago. He became a computer operator in the early 1960s; an associate professor at Carnegie Mellow University in 1963 and a full professor of computer science at Stanford University in 1969. He did pioneering work on software engineering from the 1960s, and made valuable contributions to the theory of parsing; the semantics of programming languages; program verification and methodologies for the creation of efficient and reliable software.

Mathematics and Computer Science were regarded as two completely separate disciplines in the 1960s, and software development was based on the assumption that the completed code would always contain defects. It was therefore better and more productive to write the code as quickly as possible, and to then perform debugging to find the defects. Programmers then corrected the defects, made patches and re-tested and found more defects. This continued until they could no longer find defects. Of course, there was always the danger that defects remained in the code that could give rise to software failures.

Fig. 4.5 Branch assertions in
flowcharts
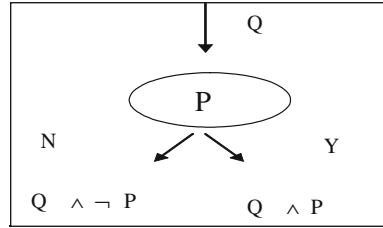
Fig. 4.6 Assignment
assertions in flowcharts
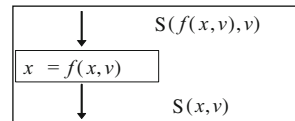
Floyd believed that there was a way to construct a rigorous proof of the correctness
of the programs using mathematics. He showed that mathematics could be used for
program verification, and he introduced the concept of *assertions* that provided a
way to verify the correctness of programs.

Flowcharts were employed in the 1960s to explain the sequence of basic steps for
computer programs. Floyd's insight was to build upon flowcharts and to apply an
invariant assertion to each branch in the flowchart. These assertions state the essential
relations that exist between the variables at that point in the flowchart. An example
relation is "$R = Z > 0, X = 1, Y = 0$". He devised a general flowchart language
to apply his method to programming languages. The language essentially contains
boxes linked by flow of control arrows [Flo:67].

Consider the assertion Q that is true on entry to a branch where the condition at
the branch is P. Then, the assertion on exit from the branch is $Q \wedge \neg P$ if P is false
and $Q \wedge P$ otherwise (see Fig. 4.5).

The use of assertions may be employed in an assignment statement. Suppose, x
represents a variable and v represents a vector consisting of all the variables in the
program. Suppose, $f(x, v)$ represents a function or expression of x and the other
program variables represented by the vector v. Suppose, the assertion $S(f(x, v), v)$
is true before the assignment $x = f(x, v)$. Then, the assertion $S(x, v)$ is true after the
assignment. This is given by Fig. 4.6.

Floyd used flowchart symbols to represent entry and exit to the flowchart. He used
entry and exit assertions to describe the program's entry and exit conditions.

Floyd's technique showed how a computer program is a sequence of logical as-
sertions. Each assertion is true whenever control passes to it, and statements appear
between the assertions. The initial assertion states the conditions that must be true for
execution of the program to take place, and the exit assertion essentially describes
what must be true when the program terminates.

His key insight was the recognition that if it can be shown that the assertion imme-
diately following each step is a consequence of the assertion immediately preceding
it, then the assertion at the end of the program will be true, provided the appropriate
assertion was true at the beginning of the program.

Fig. 4.7 C.A.R. Hoare

He published an influential paper, "Assigning Meanings to Programs", in 1967 [Flo:67], and this paper influenced Hoare's work on pre- and post-conditions leading to Hoare logic [Hor:69]. Floyd's paper also presented a formal grammar for flowcharts, together with rigorous methods for verifying the effects of basic actions such as assignments.

He also did research work on compilers and on the translation of programming languages into machine languages. This included work on the theory of parsing [Flo:63, Flo:64] and on the theory of compilers. His work led to improved algorithms for parsing sentences and phrases in programming languages. He worked closely with Donald Knuth, and reviewed Knuth's *The Art of Computer Programming*.[11] He received the Turing Award in 1978 for his influence on methodologies for the creation of efficient and reliable software, and for his contribution to the theory of parsing, the semantics of programming languages, the analysis of algorithms and for program verification.

Hoare logic is a formal system of logic used for programming semantics and for program verification. It was developed by the well-known British computer scientist, C.A.R. Hoare, and was originally published in Hoare's 1969 paper "An Axiomatic Basis for Computer Programming" [Hor:69]. Hoare and others have subsequently refined it, and it provides a logical methodology for precise reasoning about the correctness of computer programs.

Hoare's early work in computing was in the early 1960s, when he worked as a programmer at Elliott Brothers in the United Kingdom (Fig. 4.7) His first assignment was the implementation of a subset of the ALGOL 60 programming language, and this language was designed by an international committee and had a concise specification of 21 pages [Nau:60]. It gave the implementer of the language accurate and sufficient information to implement a compiler for the language, and there was no need for the

---

[11] *The Art of Computer Programming* [Knu:97] was originally published in three volumes. Volume 1 appeared in 1968; Vol. 2 in 1969 and Vol. 3 in 1973.

implementer to communicate with the language designers on the precise meaning of the language constructs.

The grammar of ALGOL 60 was specified in Backus Naur Form (BNF). The success of BNF in specifying the syntax of ALGOL 60 led to its use in the specification of the syntax of other programming languages, and to a growth in research in the theory of formal semantics of programming languages. One view that existed at that time was that a compiler and its target implementation should give the meaning of a language. Hoare argued that the meaning of a language should be independent of its implementation on the machine that the language is to be run on. He preferred to avoid operational approaches to programming language semantics, and proposed instead the axiomatic approach.

He became professor of Queens University in Belfast in 1968 and was influenced by Floyd's 1967 paper that applied assertions to flowcharts. Hoare recognised that this provided an effective method for proving the correctness of programs, and he built upon Floyd's approach to cover the familiar constructs of high-level programming languages.

This led to the *axiomatic approach* to defining the semantics of every statement in a programming language, and the approach consists of axioms and proof rules. He introduced what has become known as the Hoare triple, and this describes how the execution of a fragment of code changes the state. A Hoare triple is of the form:

$$P\{Q\}R$$

where, P and R are assertions and Q is a program or command. The predicate P is called the *pre-condition*, and the predicate R is called the *post-condition*.

Definition 4.2 (Partial Correctness) *The meaning of the Hoare triple above is that whenever the predicate P holds of the state before the execution of the command or program Q, then the predicate R will hold after the execution of Q. The brackets indicate partial correctness, as if Q does not terminate, then R can be any predicate. R may be chosen to be false to express that Q does not terminate.*

Total correctness requires Q to terminate, and at termination R is true. Termination needs to be proved separately. Hoare logic includes axioms and rules of inference rules for the constructs of imperative programming language.

Hoare and Dijkstra were of the view that the starting point of a program should always be the specification, and that the proof of the correctness of the program should be developed along with the program itself.

That is, the starting point is the mathematical specification of what a program is to do, and mathematical transformations are applied to the specification until it is turned into a program that can be executed. The resulting program is then known to be correct by construction.

4.4 Software Engineering Mathematics

The use of mathematics plays a key role in the classical engineer's work. For example, bridge designers will develop a mathematical model of a bridge prior to its construction. The model is a simplification of the reality, and an exploration of the

Table 4.1 Classical mathematics for software engineering

Area	Description
Set theory	This material is elementary but fundamental. It was discussed in Chap. 2
Relations	A relation between A and B is a subset of $A \times B$. For example, the relation $T(A, A)$ where $(a_1, a_2) \in T$ if a_1 is taller than a_2
Functions	A function $f: A \to B$ is a relation where for each $a \in A$ there is exactly one $b \in B$ such that $(a, b) \in f$. This is denoted by $f(a) = b$
Logic	Logic is the foundation for formal reasoning. It includes the study of propositional and predicate calculus. It was discussed in Chap. 3
Calculus	Calculus is used extensively in science and engineering to solve practical problems. It includes differentiation and integration, numerical methods, differential equations, etc.
Probability and statistics	Probability theory is concerned with determining the mathematical probability of various events occurring. It has been applied to the software reliability field
Finite state machines	Finite state machines are mathematical entities that are employed to model the execution of a program
Graph theory	Graphs are useful in modelling computer networks, and a graph consists of vertices and edges
Matrix theory	This includes the study of $m \times n$-dimensional matrices

model enables a deeper understanding of the proposed bridge to be gained. Engineers will model the various stresses on the bridge to ensure that the bridge design can deal with the projected traffic flow. The engineer applies mathematics and models to the design of the product, and the analysis of the design is a mathematical activity.

Mathematics allows a rigorous analysis to take place and avoids an over-reliance on intuition. The emphasis is on applied mathematics to solve practical problems and to develop products that are fit for use. Engineers are taught how to apply mathematics in their work, and the emphasis is always on the application of mathematics to solve practical problems.

Classical mathematics may be applied to software engineering, and specialised mathematical methods and notations have also been developed. These include specialised formal specification languages such as Z and VDM, and classical mathematics as in Table 4.1.

Mathematical approaches to software engineering are described in [ORg:06]. Next, we consider formal methods which may be employed in the development of high-quality software.

4.5 Formal Methods

The term "formal methods" refers to various mathematical techniques used in the software field for the specification and formal development of software. Formal methods consist of formal specification languages or notations, and employ a collection of tools to support the syntax checking of the specification as well as the proof of properties about the specification.

The mathematical analysis of the formal specification allows questions to be asked about what the system does, and these questions may be answered independently of the implementation. Mathematical notation is precise, and this helps to avoid the problem of ambiguity inherent in a natural language description of a system.

The formal specification may be used to promote a common understanding for all stakeholders, and it becomes the key reference point for the project team in their individual activities such as requirements gathering; design and development as well as testing and program documentation. The term "formal methods" is used to describe a formal specification language and a method for the design and implementation of computer systems.

Formal methods have been applied to a diverse range of applications, including the safety-critical field; security-critical field; the railway sector; the nuclear field; microprocessor verification; the specification of standards and the specification and verification of programs.

There is a strong motivation to use best practice in software engineering in order to produce software adhering to high quality standards. Many companies employ best in class software engineering processes, and formal methods are one leading-edge technology that may assist companies in reducing the occurrence of defects in their software products.

There are various tools to support formal methods including syntax checkers that determine whether the specification is syntactically correct; specialised editors to assist in editing to ensure that the written specification is syntactically correct; tools to support refinement; automated code generators to generate a high-level language corresponding to the specification; theorem provers to demonstrate the presence or absence of key properties and to prove the correctness of refinement steps, and to identify and resolve proof obligations and specification animation tools where the execution of the specification can be simulated.

Formal methods have been mainly applied to the safety-critical and security-critical fields. They need to mature further before they will be used in mainstram software engineering. They are described in more detail in Chap. 5.

4.6   Software Inspections and Testing

Software inspections play an important role in building quality into software products. The Fagan Inspection Methodology was developed by Michael Fagan at IBM in the mid-1970s [Fag:76]. It is a seven-step process that identifies and removes defects in work products. There is a strong economic case for identifying defects as early as possible, as the cost of their correction increases the later that they are discovered in the lifecycle. The Fagan methodology mandates that requirement documents, design documents, source code and test plans are all formally inspected.

There are several *roles* defined in the process including the *moderator* who chairs the inspection; the *reader* who reads or paraphrases the particular deliverable; the *author* who is the creator of the deliverable and the *tester* who is concerned with the testing viewpoint.

The inspection process will consider whether a design is correct with respect to the requirements, and whether the source code is correct with respect to the design. There are seven stages in the process [ORg:02]:

- Planning.
- Overview.
- Prepare.
- Inspect.
- Process improvement.
- Re-work.
- Follow-up.

The defects identified may be classified into various types and mature organisations record the inspection data in a database for further analysis. Metrics may be employed to determine the effectiveness of the organisation in identifying errors in phase, and detecting defects out of phase. Tom Gilb has defined an alternate inspection methodology [Glb:94].

Software testing plays a key role in verifying that a software product is of high quality and conforms to the customer's quality expectations. Testing is both a constructive activity in that it is verifying the correctness of functionality, and it is also a destructive activity in that the objective is to find as many defects as possible in the software. The testing verifies that the requirements are correctly implemented as well as identifying whether any defects are present in the software product.

There are various types of testing such as unit testing, integration testing, system testing, performance testing, usability testing, regression testing and customer acceptance testing. The testing needs to be planned and test cases prepared and executed. The results of testing are reported and any issues corrected and re-tested. The test cases will need to be appropriate to verify the correctness of the software. The quality of the testing is dependent on the maturity of the test process, and a good test process will include:

- Test planning and risk management.
- Dedicated test environment and test tools.
- Test case definition.
- Test automation.
- Formality in handover to test department.
- Test execution.
- Test result analysis.
- Test reporting.
- Measurements of test effectiveness.
- Post mortem and test process improvement.

Metrics are generally maintained to provide visibility into the effectiveness of the testing process. Software inspection and testing are described in more detail in [ORg:02, ORg:10].

Fig. 4.8 Watts Humphrey.
(Courtesy of Watts
Humphrey)

4.7   Process Maturity Models

The SEI developed the CMM in the early 1990s as a framework to help software organisations to improve their software process maturity and to implement best practice in software and systems engineering. The SEI believes that there is a close relationship between the maturity of software processes and the quality of the delivered software product.

The CMM applied the ideas of Deming [Dem:86], Juran [Jur:00] and Crosby [Crs:79] to the software field. These quality gurus were influential in transforming manufacturing companies with quality problems to effective quality driven organisations with a reduced cost of poor quality.

They recognised the need to focus on process improvement, and Watt Humphries did early work on software process improvement at IBM [Hum:89]. He moved to the SEI in the late 1980s and the first version of the CMM was released in 1991 (Fig. 4.8). It is now called the Capability Maturity Model Integration (CMMI®) [CKS:11].

It consists of five maturity levels with each maturity level (except level 1) consisting of several process areas. Each process area consists of a set of goals that are implemented by practices related to that process area leading to an effective process.

The emphasis on level 2 of the CMMI is on maturing management practices such as project management, requirements management, configuration management and so on. The emphasis on level 3 of the CMMI is to mature engineering and organisation practices. This maturity level includes peer reviews and testing, requirements development, software design and implementation practices and so on. Level 4 is concerned with ensuring that key processes are performing within strict quantitative limits, and adjusting processes, where necessary, to perform within these defined limits. Level 5 is concerned with continuous process improvement, which is quantitatively verified.

Maturity levels may not be skipped in the staged implementation of the CMMI. There is also a continuous representation of the CMMI, which allows the organisation to focus on the improvements to key processes. However, in practice, it is often necessary to implement several of the level 2 process areas before serious work can be done on implementing a process at a higher maturity level. The use of metrics

[Fen:95, Glb:76] becomes more important as an organisation matures, as metrics allow the performance of an organisation to be objectively judged. The higher CMMI maturity levels set quantitative levels for processes to perform within.

The CMMI allows organisations to benchmark themselves against other similar organisations. This is done by the formal SEI-approved Standard CMMI Appraisal Method for Process Improvement (SCAMPI) appraisals conducted by an authorised SCAMPI lead appraiser. The results of a SCAMPI appraisal are generally reported back to the SEI, and there is a strict qualification process to become an authorised lead appraiser. An appraisal is useful in verifying that an organisation has improved, and it enables the organisation to prioritise improvements for the next improvement cycle.

The time required to implement the CMMI in an organisation depends on the current maturity and size of the organisation. It generally takes 1–2 years to implement maturity level 2, and a further 1–2 years to implement level 3.

4.8   Review Questions

1. What is software engineering and describe the difference between classical engineers and software engineers.
2. Describe the "software crisis" of the late 1960s that led to the first software engineering conference in 1968.
3. Discuss the Standish Research Report and the level of success of IT projects today. In your view, is there a crisis in software engineering today? Give reasons for your answer.
4. Discuss what the role of mathematics should be in current software engineering.
5. Describe the waterfall and spiral lifecycles. What are the similarities and differences between them?
6. Discuss the contributions of Floyd and Hoare.
7. Explain the difference between partial correctness and total correctness.
8. What are formal methods?
9. Discuss the process maturity models (including the CMMI). What are their advantages and disadvantages?
10. Discuss how software inspections and testing can assist in the delivery of high-quality software.

4.9   Summary

This chapter considered a short history of some important developments in software engineering. Its birth was at the Garmisch conference in 1968, and it was recognised that there was a crisis in the software field, and a need for

sound methodologies to design, develop and maintain software to meet customer needs.

Classical engineering has a successful track record in building high-quality products that are safe for the public to use. It is therefore natural to consider using an engineering approach to developing software, and this involves identifying the customer requirements, carrying out a rigorous design to meet the requirements, developing and coding a solution to meet the design, and conducting appropriate inspections and testing to verify the correctness of the solution.

Mathematics plays a key role in engineering to assist with design and verification of software products. It is therefore reasonable to apply appropriate mathematics in software engineering (especially for safety-critical systems) to assure that the delivered systems conform to the requirements. The extent to which mathematics should be used is controversial with strong views in both camps. In many cases, peer reviews and testing will be sufficient to build quality into the software product. In other cases, and especially with safety and security-critical applications, it is desirable to have the extra assurance that may be provided by mathematical techniques.

There is a lot more to the successful delivery of a project than just the use of mathematics or peer reviews and testing. Sound project management and quality management practices are essential, as a project that is not properly managed will suffer from schedule, budget or cost overruns as well as problems with quality.

Maturity models such as the CMMI can assist organisations in maturing key management and engineering practices, and may help companies in their goals to deliver high-quality software systems that are consistently delivered on time and budget.

Chapter 5
Formal Methods

Key Topics

Vienna Development Method
Z Specification Language
B-Method
Process Calculus
Finite State Machines
Model-oriented approach
Axiomatic approach
Usability of Formal Methods

5.1 Introduction

The term "formal methods" refer to various mathematical techniques used for the formal specification and development of software. They consist of a formal specification language, and employ a collection of tools to support the syntax checking of the specification as well as the proof of properties of the specification. They allow questions to be asked about what the system does independently of the implementation.

The use of mathematical notation avoids speculation about the meaning of phrases in an imprecisely worded natural language description of a system. Natural language is inherently ambiguous, whereas mathematics employs a precise rigorous notation. Spivey [Spi:92] defines formal specification as:

Definition 5.1 (Formal Specification) *Formal specification is the use of mathematical notation to describe in a precise way the properties that an information system must have, without unduly constraining the way in which these properties are achieved.*

The formal specification thus becomes the key reference point for the different parties involved in the construction of the system. It may be used as the reference point in the requirements; program implementation and testing and program documentation. It promotes a common understanding for all those concerned with the system.

The term "formal methods" is used to describe a formal specification language and a method for the design and implementation of computer systems.

The specification is written in a mathematical language, and the implementation may be derived from the specification via step-wise refinement.[1] The refinement step makes the specification more concrete and closer to the actual implementation. There is an associated proof obligation to demonstrate that the refinement is valid, and that the concrete state preserves the properties of the more abstract state. Thus, assuming that the original specification is correct and the proofs of correctness of each refinement step are valid, then there is a very high degree of confidence in the correctness of the implemented software. Step-wise refinement is illustrated as follows: the initial specification S is the initial model M_0; it is then refined into the more concrete model M_1, and M_1 is then refined into M_2 and so on until the eventual implementation $M_n = E$ is produced.

$$S = M_0 \subseteq M_1 \subseteq M_2 \subseteq M_3 \subseteq \cdots \subseteq M_n = \mathrm{E}$$

Requirements are the foundation of the system to be built, and irrespective of the best design and development practices, the product will be incorrect if the requirements are incorrect. The objective of requirements validation is to ensure that the requirements reflect what is actually required by the customer (in order to build the right system). Formal methods may be employed to model the requirements, and the model exploration yields further desirable or undesirable properties. The ability to prove that certain properties are true of the specification is very valuable, especially in safety-critical and security-critical applications. These properties are logical consequences of the definition of the requirements, and, where appropriate, the requirements may be amended. Thus, formal methods may be employed in a sense to debug the requirements during requirements validation.

The use of formal methods generally leads to more robust software and to increased confidence in its correctness. The challenges involved in the deployment of formal methods in an organisation include the education of staff in formal specification, as the use of these mathematical techniques may be a culture shock to many staff.

Formal methods have been applied to a diverse range of applications, including the security-critical field; the safety-critical field; the railway sector; microprocessor verification; the specification of standards and the specification and verification of programmes.

Parnas and others have criticised formal methods on the grounds described in Table 5.1.

However, formal methods are potentially quite useful and reasonably easy to use. The use of a formal method such as Z or Vienna Development Method (VDM)

---

[1] It is questionable whether step-wise refinement is cost effective in mainstream software engineering, as it involves re-writing a specification *ad nauseum*. It is time-consuming to proceed in refinement steps with significant time also required to prove that the refinement step is valid. It is more relevant to the safety-critical field. Others in the formal methods field may disagree with this position.

Table 5.1 Criticisms of formal methods

No.	Criticism
1	Often the formal specification is as difficult to read as the program[a]
2	Many formal specifications are wrong[b]
3	Formal methods are strong on syntax but provide little assistance in deciding on what technical information should be recorded using the syntax[c]
4	Formal specifications provide a model of the proposed system. However, a precise unambiguous mathematical statement of the requirements is what is needed[d]
5	Step-wise refinement is unrealistic.[e] It is like, for example, deriving a bridge from the description of a river and the expected traffic on the bridge. There is always a need for a creative step in design
6	Much unnecessary mathematical formalisms have been developed rather than using the available classical mathematics[f]

[a]Of course, others might reply by saying that some of Parnas's tables are not exactly intuitive, and that the notation he employs in some of his tables is quite unfriendly. The usability of all of the mathematical approaches needs to be enhanced if they are to be taken seriously by industrialists

[b]Obviously, the formal specification must be analysed using mathematical reasoning and tools to provide confidence in its correctness. The validation of a formal specification can be carried out using mathematical proof of key properties of the specification; software inspections or specification animation

[c]Approaches such as Vienna Development Method (VDM) include a method for software development as well as the specification language

[d]Models are extremely valuable as they allow simplification of the reality. A mathematical study of the model demonstrates whether it is a suitable representation of the system. Models allow properties of the proposed requirements to be studied prior to the implementation

[e]Step-wise refinement involves rewriting a specification with each refinement step producing a more concrete specification (that includes code and formal specification) until eventually the detailed code is produced. However, tool support may make refinement easier

[f]Approaches such as VDM or Z are useful in such a way that they add greater rigour to the software development process. They are reasonably easy to learn, and there have been some good results obtained by their use. Classical mathematics is familiar to students and therefore it is desirable that new formalisms are introduced only where absolutely necessary

forces the software engineer to be precise and helps to avoid ambiguities present in natural language. Clearly, a formal specification should be subject to peer review to provide confidence in its correctness. New formalisms need to be intuitive to be usable by practitioners. The advantage of classical mathematics is that it is familiar to students.

5.2 Why Should We Use Formal Methods?

There is a strong motivation to use best practice in software engineering in order to produce software adhering to high quality standards. Quality problems with software may cause minor irritations or major damage to a customer's business including loss of life. Formal methods are a leading-edge technology that may be of benefit to companies in reducing the occurrence of defects in software products. Brown [Bro:90] argues that for the safety-critical field that:

Comment 5.1 (Missile Safety) *Missile systems must be presumed dangerous until shown to be safe, and that the absence of evidence for the existence of dangerous errors does not amount to evidence for the absence of danger.*

This suggests that companies will need to demonstrate that every reasonable practice was taken to prevent the occurrence of defects. One such practice is the use of formal methods, and its exclusion may need to be justified in some domains. It is quite possible that a software company may be sued for a software which injures a third party, and this suggests that companies will need a rigorous quality-assurance system to prevent the occurrence of defects.

There is some evidence to suggest that the use of formal methods provides savings in the cost of the project. For example, a 9 % cost saving is attributed to the use of formal methods during the Customer Information Control System (CICS) project; the T800 project attributes a 12-month reduction in testing time to the use of formal methods. These are discussed in more detail in Chap. 1 of [HB:95].

The use of formal methods is mandatory in certain circumstances. The Ministry of Defence in the United Kingdom issued two safety-critical standards[2] in the early 1990s related to the use of formal methods in the software development lifecycle.

The first is Defence Standard 00-55, "The Procurement of safety-critical software in defense equipment" [MOD:91a], which makes it mandatory to employ formal methods in safety-critical software development in the United Kingdom; and mandates the use of formal proof that the most crucial programmes correctly implement their specifications.

The other is Defence Standard 00-56 "Hazard analysis and safety classification of the computer and programmable electronic system elements of defense equipment" [MOD:91b]. The objective of this standard is to provide guidance to identify which systems or parts of systems being developed are safety critical and thereby require the use of formal methods. This proposed system is subject to an initial hazard analysis to determine whether there are safety-critical parts.

The reaction to these defence standards 00-55 and 00-56 was quite hostile initially, as most suppliers were unlikely to meet the technical and organisation requirements of the standard. This is described in [Tie:91].

5.3   Applications of Formal Methods

Formal methods have been employed to verify correctness in the nuclear power industry, the aerospace industry, the security technology area and the railroad domain. These sectors are subject to stringent regulatory controls to ensure safety and security. Several organisations have piloted formal methods with varying degrees of success. These include IBM, who developed VDM at its laboratory in Vienna; IBM (Hursley) piloted the Z formal specification language on the CICS project.

---

[2] The U.K. Defence Standards 00-55 and 00-56 have been revised in recent years to be less prescriptive on the use of formal methods.

The mathematical techniques developed by Parnas (i.e. requirements model and tabular expressions) have been employed to specify the requirements of the A-7 aircraft as part of a research project for the US Navy.[3] Tabular expressions have also been employed for the software inspection of the automated shutdown software of the Darlington Nuclear power plant in Canada.[4] These are two successful uses of mathematical techniques in software engineering.

There are examples of the use of formal methods in the railway domain, and examples dealing with the modelling and verification of a railroad gate controller and railway signalling are described in [HB:95]. Clearly, it is essential to verify safety-critical properties such as "when the train goes through the level crossing then the gate is closed".

5.4   Tools for Formal Methods

A key criticism of formal methods is the limited availability of tools to support the software engineer in writing the formal specification and in conducting proof. Many of the early tools were criticised as not being of industrial strength. However, in recent years, more advanced tools to support the software engineer's work in formal specification and formal proof have become available, and this is likely to continue in the coming years.

The tools include syntax checkers that determine whether the specification is syntactically correct; specialised editors which ensure that the written specification is syntactically correct; tools to support refinement; automated code generators that generate a high-level language corresponding to the specification; theorem provers to demonstrate the presence or absence of key properties and to prove the correctness of refinement steps, and to identify and resolve proof obligations and specification animation tools where the execution of the specification can be simulated.

The B-Toolkit from B-Core is an integrated set of tools that supports the B-Method. These include syntax and type checking, specification animation, proof obligation generator, an auto-prover, a proof assistor and code generation. This allows, in theory, a complete formal development from initial specification to final implementation to be achieved, with every proof obligation justified, leading to a provably correct program.

The IFAD Toolbox[5] is a support tool for the VDM-SL specification language, and it includes support for syntax and type checking, an interpreter and debugger to execute and debug the specification and a code generator to convert from VDM-SL

---

[3] However, the resulting software was never actually deployed on the A-7 aircraft.

[4] This was an impressive use of mathematical techniques and it has been acknowledged that formal methods must play an important role in future developments at Darlington. However, given the time and cost involved in the software inspection of the shutdown software some managers have less enthusiasm in shifting from hardware to software controllers [Ger:94].

[5] The IFAD Toolbox has been renamed to VDMTools as IFAD sold the VDM Tools to CSK in Japan. The tools are expected to be available worldwide and will be improved further.

to C++. It also includes support for graphical notations such as the object-modelling technique (OMT)/unified-modelling language UML design notations.

5.5 Approaches to Formal Methods

There are two key approaches to formal methods: namely the *model-oriented approach* of VDM or Z, and the *algebraic* or *axiomatic approach* of the process calculi such as the calculus communicating systems (CCS) or communicating sequential processes (CSP).

5.5.1 Model-Oriented Approach

The model-oriented approach to specification is based on mathematical models, and a model is a mathematical representation or abstraction of a physical entity or system. The model aims to provide a mathematical explanation of the behaviour of the physical world, and it is considered suitable if its properties closely match those of the system, and if its calculations match and simplify calculations in the real world. A model will allow predictions of future behaviour to be made. There are many models employed in the physical world such as models of the weather system that allow weather predictions to be made.

It is fundamental to explore the model to determine its adequacy, the extent to which it explains the underlying physical behaviour and allows predictions of future behaviour to be made. This will determine its acceptability as a representation of the physical world. Models that are ineffective will be replaced with models that offer a better explanation of the manifested physical behaviour. There are many examples in science of the replacement of one theory by a newer one. For example, the Copernican model of the universe replaced the older Ptolemaic model, and Newtonian physics was replaced by Einstein's theories on relativity. The structure of the revolutions that take place in science are described in [Kuh:70].

The model-oriented approach to software development involves defining an abstract model of the proposed software system. The model acts as a representation of the proposed system, and the model is then explored to assess its suitability. The exploration of the model takes the form of model interrogation, i.e. asking questions and determining the effectiveness of the model in answering the questions. The modelling in formal methods is typically performed via elementary discrete mathematics, including set theory, sequences, functions and relations.

VDM and Z are model-oriented approaches to formal methods. VDM arose from work done in the IBM laboratory in Vienna in formalising the semantics for the PL/1 compiler, and it was later applied to the specification of software systems. The origin of the Z specification language lies in the work done at Oxford University in the early 1980s.

5.5.2 Axiomatic Approach

The axiomatic approach focuses on the properties that the proposed system is to satisfy, and there is no intention to produce a model of the system. The required properties and behaviour of the system are stated in mathematical notation. The difference between the axiomatic specification and a model-based approach can be seen in the example of a stack.

The stack includes operators for pushing an element onto the stack and popping an element from the stack. The properties of *pop* and *push* are explicitly defined in the axiomatic approach. The model-oriented approach constructs an explicit model of the stack and the operations are defined in terms of the effect that they have on the model. The specification of the *pop* operation on a stack is given by axiomatic properties, for example, $pop(push(s, x)) = s$.

Comment 5.2 (Axiomatic Approach) *The property-oriented approach has the advantage that the implementer is not constrained to a particular choice of implementation, and the only constraint is that the implementation must satisfy the stipulated properties.*

The emphasis is on the required properties of the system, and implementation issues are avoided. The focus is on the specification of the underlying behaviour, and properties are typically stated using mathematical logic or higher order logics. Mechanised theorem-proving techniques may be employed to prove results.

One potential problem with the axiomatic approach is that the properties specified may not be satisfiable in any implementation. Thus, whenever a "formal axiomatic theory" is developed a corresponding "model" of the theory must be identified, in order to ensure that the properties may be realised in practice. That is, when proposing a system that is to satisfy some set of properties, there is a need to prove that there is at least one system that will satisfy the set of properties.

5.6 Proof and Formal Methods

The word "proof" has several connotations in various disciplines; for example, in a court of law, the defendant is assumed innocent until proven guilty. The proof of the guilt of the defendant may take the form of certain facts in relation to the movements of the defendant, the defendant's circumstances, the defendant's alibi, statements taken from witnesses, rebuttal arguments from the defence and certain theories produced by the prosecution or defence. Ultimately, in the case of a trial by the jury, the defendant is judged guilty or not guilty depending on the extent to which the jury has been convinced by the arguments made by the prosecution and defence.

A mathematical proof typically includes natural language and mathematical symbols, and often many of the tedious details of the proof are omitted. The proof may employ a "divide and conquer" technique; i.e. breaking the conjecture down into

subgoals and then attempting to prove the subgoals. Many proofs in formal methods are concerned with crosschecking the details of the specification or checking the validity of refinement steps, or checking that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and theorem provers[6] are essential in assisting with this. Machine proof needs to be explicit, and reliance on some brilliant insight is avoided. Proofs by hand are notorious for containing errors or jumps in reasoning, while machine proofs are explicit but are often extremely lengthy and unreadable (e.g. the actual machine proof of correctness of the VIPER microprocessor[7] [Tie:91] consisted of several million formulae).

A formal mathematical proof consists of a sequence of formulae, where each element is either an axiom or derived from a previous element in the series by applying a fixed set of mechanical rules.

Theorem provers are invaluable in resolving many of the thousands of proof obligations that arise from a formal specification, and the application of formal methods in an industrial environment requires the use of machine-assisted proof. Automated theorem proving is difficult, as often mathematicians prove a theorem with an initial intuitive feeling that the theorem is true. Human intervention to provide guidance or intuition improves the effectiveness of the theorem prover.

The proof of various properties about a program increases confidence in its correctness. However, an absolute proof of correctness[8] is unlikely except for the most trivial of programmes. A program may consist of legacy software that is assumed to work; a compiler that is assumed to work correctly creates it. Theorem provers are programmes that are assumed to function correctly. The best that formal methods can claim is increased confidence in correctness of the software, rather than an absolute proof of correctness.

5.7   The Future of Formal Methods

The debate concerning the level of use of mathematics in software engineering is still ongoing. Many practitioners are against the use of mathematics and avoid its use. They tend to employ methodologies such as software inspections and testing to improve confidence in the correctness of the software. They argue that in the current competitive industrial environment where time to market is a key driver that the use of such formal mathematical techniques would seriously impact the market opportunity.

---

[6] Most existing theorem provers are difficult to use and are for specialist use only. There is a need to improve the usability of theorem provers.

[7] This verification was controversial with RSRE and Charter overselling VIPER as a chip design that conforms to its formal specification.

[8] This position is controversial with others arguing that if correctness is defined mathematically then the mathematical definition (i.e. formal specification) is a theorem, and the task is to prove that the program satisfies the theorem. They argue that the proofs for non-trivial programs exist, and that the reason why there are not many examples of such proofs is due to a lack of mathematical specifications.

Industrialists often need to balance conflicting needs such as quality, cost and on-time delivery. They argue that the commercial necessities require methodologies and techniques that allow them to achieve their business goals effectively.

The other camp argues that the use of mathematics is essential in the delivery of high-quality and reliable software, and that if a company does not place sufficient emphasis on quality it will pay the price in terms of poor quality and its reputation in the market place.

It is generally accepted that mathematics and formal methods must play a role in the safety-critical and security-critical fields. Apart from that the extent of the use of mathematics is a hotly disputed topic. The pace of change in the world is extraordinary, and companies face competitive forces in a global market place. It is unrealistic to expect companies to deploy formal methods unless they have clear evidence that it will support them in delivering commercial products to the market place ahead of their competition, at the right price and with the right quality. Formal methods need to prove that it can do this if it wishes to be taken seriously in main-stream software engineering. The issue of technology transfer of formal methods to industry is discussed in [ORg:06].

5.8   The Vienna Development Method

VDM dates from work done by the IBM research laboratory in Vienna. This group was specifying the semantics of the PL/1 programming language using an opera-tional semantic approach. That is, the semantics of the language were defined in terms of a hypothetical machine which interprets the programmes of that language [BjJ:78, BjJ:82]. Later work led to the VDM with its specification language, Meta IV. This was used to give the denotational semantics of programming languages; i.e. a mathematical object (set, function, etc.) is associated with each phrase of the language [BjJ:82]. The mathematical object is termed the *denotation* of the phrase.

VDM is a *model-oriented approach* and this means that an explicit model of the state of an abstract machine is given, and operations are defined in terms of this state. Operations may act on the system state, taking inputs, and producing outputs as well as a new system state. Operations are defined in a pre-condition and post-condition style. Each operation has an associated proof obligation to ensure that if the pre-condition is true, then the operation preserves the system invariant. The initial state itself is, of course, required to satisfy the system invariant.

VDM uses keywords to distinguish different parts of the specification, e.g. pre-conditions, post-conditions, as introduced by the keywords *pre* and *post*, respectively. In keeping with the philosophy that formal methods specifies *what* a system does as distinct from *how*, VDM employs post-conditions to stipulate the effect of the operation on the state. The previous state is then distinguished by employing *hooked variables*, e.g. v⁻, and the post-condition specifies the new state which is defined by a logical predicate relating the pre-state to the post-state.

VDM is more than its specification language VDM-SL, and is, in fact, a software development method, with rules to verify the steps of development. The rules enable the executable specification, i.e. the detailed code, to be obtained from the initial specification via refinement steps. Thus, we have a sequence $S = S_0, S_1, \ldots, S_n = E$ of specifications, where S is the initial specification, and E is the final (executable) specification.

Retrieval functions enable a return from a more concrete specification to the more abstract specification. The initial specification consists of an initial state, a system state, and a set of operations. The system state is a particular domain, where a domain is built out of primitive domains such as the set of natural numbers, etc. or constructed from primitive domains using domain constructors such as Cartesian product, disjoint union, etc. A domain-invariant predicate may further constrain the domain, and a *type* in VDM reflects a domain obtained in this way. Thus, a type in VDM is more specific than the signature of the type, and thus represents values in the domain defined by the signature, which satisfy the domain invariant. In view of this approach to types, it is clear that VDM types may not be "statically type checked".

VDM specifications are structured into modules, with a module containing the module name, parameters, types, operations, etc. Partial functions occur frequently in computer science as many functions, may be undefined, or fail to terminate for some arguments in their domain. VDM addresses partial functions by employing nonstandard logical operators, namely the logic of partial functions (LPFs) discussed in Chap. 3.

VDM has been used in industrial projects, and its tool support includes the IFAD Toolbox.[9] VDM is described in more detail in [ORg:06]. There are several variants of VDM, including VDM^{++}, the object-oriented extension of VDM, and the Irish school of the VDM, which is discussed in the next section.

5.9   VDM♣, the Irish School of Vienna Development Method (VDM)

The Irish School of VDM is a variant of standard VDM, and is characterised by its constructive approach, classical mathematical style and terse notation [Mac:90]. This method aims to combine the *what* and *how* of formal methods in that its terse specification style stipulates in concise form *what* the system should do; furthermore, the fact that its specifications are constructive (or functional) means that the *how* is included with the *what*. However, it is important to qualify this by stating that the how as presented by VDM♣ is not directly executable, as several of its mathematical data types have no corresponding structure in high-level programming languages or functional languages. Thus, a conversion or reification of the specification into a functional or higher level language must take place to ensure a successful execution.

---

[9] The VDM Tools are now available from the CSK Group in Japan.

Further, the fact that a specification is constructive is no guarantee that it is a good implementation strategy, if the construction itself is naive.

The Irish school follows a similar development methodology as in standard VDM, and is a model-oriented approach. The initial specification is presented, with initial state and operations defined. The operations are presented with pre-conditions; however, no post-condition is necessary as the operation is "functionally" (i.e. explicitly) constructed.

There are proof obligations to demonstrate that the operations preserve the invariant. That is, if the pre-condition for the operation is true, and the operation is performed, then the system invariant remains true after the operation. The philosophy is to exhibit existence *constructively* rather than a theoretical proof of existence that demonstrates the existence of a solution without presenting an algorithm to construct the solution.

The school avoids the existential quantifier of predicate calculus and reliance on logic in proof is kept to a minimum, and emphasis instead is placed on equational reasoning. Structures with nice algebraic properties are sought, and one nice algebraic structure employed is the monoid, which has closure, associativity, and a unit element. The concept of isomorphism is powerful, reflecting that two structures are essentially identical, and thus we may choose to work with either, depending on which is more convenient for the task in hand.

The school has been influenced by the work of Polya and Lakatos. The former [Pol:57] advocated a style of problem solving characterised by first considering an easier sub-problem, and considering several examples. This generally leads to a clearer insight into solving the main problem. Lakatos's [Lak:76] approach to mathematical discovery is characterised by heuristic methods. A primitive conjecture is proposed and if global counter-examples to the statement of the conjecture are discovered, then the corresponding *hidden lemma* for which this global counterexample is a local counter example is identified and added to the statement of the primitive conjecture. The process repeats, until no more global counterexamples are found. A sceptical view of absolute truth or certainty is inherent in this.

Partial functions are the norm in VDM♣, and as in standard VDM, the problem is that functions may be undefined, or fail to terminate for several of the arguments in their domain. The LPFs is avoided, and instead care is taken with recursive definitions to ensure termination is achieved for each argument. Academic and industrial projects have been conducted using the method of the Irish school, but at this stage tool support is limited.

5.10   The Z Specification Language

Z is a formal specification language founded on Zermelo set theory, and Abrial developed it at Oxford University in the early 1980s. It is used for the formal specification of software and is a model-oriented approach. An explicit model of the state of an abstract machine is given, and the operations are defined in terms of the effect on

the state. It includes a mathematical notation that is similar to VDM, and the visually striking schema calculus, which consists essentially of boxes, with these boxes or schemas used to describe operations and states. The schema calculus enables schemas to be used as building blocks and combined with other schemas. The Z specification language was published as an ISO standard (ISO/IEC 13568:2002) in 2002.

The schema calculus is a powerful means of decomposing a specification into smaller pieces or schemas. This helps to make Z specification highly readable, as each individual schema is small in size and self-contained. The exception handling is done by defining schemas for the exception cases, and these are then combined with the original operation schema. Mathematical data types are used to model the data in a system and these data types obey mathematical laws. These laws enable simplification of expressions and are useful with proofs.

Operations are defined in a pre-condition /post-condition style. However, the pre-condition is implicitly defined within the operation; i.e. it is not separated out as in standard VDM. Each operation has an associated proof obligation to ensure that if the pre-condition is true, then the operation preserves the system invariant. The initial state itself is, of course, required to satisfy the system invariant. Post-conditions employ a logical predicate which relates the pre-state to the post-state, and the post-state of a variable v is given by priming, e.g. v'. Various conventions are employed, e.g. $v?$ indicates that v is an input variable and $v!$ indicates that v is an output variable. The symbol Ξ Op operation indicates that this operation does not affect the state, whereas Δ Op indicates that this operation that affects the state.

Many data types employed in Z have no counterpart in standard programming languages. It is therefore important to identify and describe the concrete data structures that will ultimately represent the abstract mathematical structures. The operations on the abstract data structures may need to be refined to yield operations on the concrete data structure that yield equivalent results. For simple systems, direct refinement (i.e. one step from abstract specification to implementation) may be possible; in more complex systems, deferred refinement is employed, where a sequence of increasingly concrete specifications are produced to yield the executable specification eventually.

Z has been successfully applied in industry, and one of its well-known successes is the CICS project at IBM Hursley in England. Z is described in more detail in Chap. 6.

5.11   The B-Method

The *B-Technologies* [McD:94] consist of three components: a method for software development, namely the B-Method; a supporting set of tools, namely, the B-Toolkit; and a generic program for symbol manipulation, namely, the B-Tool (from which the B-Toolkit is derived). The B-Method is a model-oriented approach and is closely related to the Z specification language. Abrial developed the B specification language, and every construct in the language has a set-theoretic counterpart,

and the method is founded on Zermelo set theory. Each operation has an explicit pre-condition.

One key purpose [McD:94] of the *abstract machine* in the B-Method is to provide encapsulation of variables representing the state of the machine and operations which manipulate the state. Machines may refer to other machines, and a machine may be introduced as a refinement of another machine. The abstract machines are specification machines, refinement machines, or implementable machines. The B-Method adopts a layered approach to design where the design is gradually made more concrete by a sequence of design layers. Each design layer is a refinement that involves a more detailed implementation in terms of abstract machines of the previous layer. The design refinement ends when the final layer is implemented purely in terms of library machines. Any refinement of a machine by another has associated proof obligations, and proof is required to verify the validity of the refinement step.

Specification animation of the Abstract Machine Notation (AMN) specification is possible with the B-Toolkit, and this enables typical usage scenarios of the AMN specification to be explored for requirements validation. This is, in effect, an early form of testing, and it may be used to demonstrate the presence or absence of desirable or undesirable behaviour. Verification takes the form of a proof to demonstrate that the invariant is preserved when the operation is executed within its pre-condition, and this is performed on the AMN specification with the B-Toolkit.

The B-Toolkit provides several tools that support the B-Method, and these include syntax and type checking; specification animation, proof obligation generator, auto prover, proof assistant and code generation. Thus, in theory, a complete formal development from initial specification to final implementation may be achieved, with every proof obligation justified, leading to a provably correct program.

The B-Method and toolkit have been successfully applied in industrial applications, including the CICS project at IBM Hursley in the United Kingdom. The automated support provided has been cited as a major benefit of the application of the B-Method and the B-Toolkit.

5.12   Predicate Transformers and Weakest Pre-Conditions

The pre-condition of a program S is a predicate, i.e. a statement that may be true or false, and it is usually required to prove that if the pre-condition Q is true: i.e. $\{Q\}S\{R\}$, then execution of S is guaranteed to terminate in a finite amount of time in a state satisfying R.

The weakest pre-condition (cf. p. 109 of [Gri:81]) of a command S with respect to a post-condition R represents the set of all states such that if execution begins in any one of these states, then execution will terminate in a finite amount of time in a state with R true. These set of states may be represented by a predicate Q', so that $wp(S, R) = wp_S(R) = Q'$, and so wp_S is a predicate transformer, i.e. it may be regarded as a function on predicates. The weakest pre-condition is the pre-condition that places the fewest constraints on the state than all of the other pre-conditions of (S,R). That is, all of the other pre-conditions are stronger than the weakest pre-condition.

The notation $Q\{S\}R$ is used to denote partial correctness and indicates that if execution of S commences in any state satisfying Q, and if execution terminates, then the final state will satisfy R. Often, a predicate Q which is stronger than the weakest pre-condition $wp(S,R)$ is employed, especially where the calculation of the weakest pre-condition is nontrivial. Thus, a stronger predicate Q such that $Q \Rightarrow wp(S, R)$ is often employed.

There are many properties associated with the weakest pre-conditions, and these may be used to simplify expressions involving weakest pre-conditions, and in determining the weakest pre-conditions of various program commands such as assignments, iterations, etc. Weakest pre-conditions may be used in developing a proof of correctness of a program in parallel with its development [ORg:06].

An imperative program may be regarded as a predicate transformer. This is since a predicate P characterises the set of states in which the predicate P is true, and an imperative program may be regarded as a binary relation on states, which may be extended to a function F, leading to the Hoare triple $P\{F\}Q$. That is, the program F acts as a predicate transformer with the predicate P regarded as an input assertion, i.e. a Boolean expression that must be true before the program F is executed, and the predicate Q is the output assertion, which is true if the program F terminates (where F commenced in a state satisfying P).

5.13   The Process Calculi

The objectives of the process calculi [Hor:85] are to provide mathematical models which provide insight into the diverse issues involved in the specification, design and implementation of computer systems which continuously act and interact with their environment. These systems may be decomposed into sub-systems that interact with each other and their environment.

The basic building block is the *process*, which is a mathematical abstraction of the interactions between a system and its environment. A process that lasts indefinitely may be specified recursively. Processes may be assembled into systems; they may execute concurrently or communicate with each other. Process communication may be synchronised, and this takes the form of a process outputting a message simultaneously to another process inputting a message. Resources may be shared among several processes. Process calculi such as CSP [Hor:85] and CCS [Mil:89] have been developed to enrich the understanding of communication and concurrency, and these calculi obey a rich collection of mathematical laws.

The expression $(a ? P)$ in CSP describes a process which first engages in event a, and then behaves as process P. A recursive definition is written as $(\mu X) \bullet F(X)$ and an example of a simple chocolate vending machine is:

$$\text{VMS} = \mu X \colon \{\text{coin, choc }\} \bullet (\text{coin ? (choc ? } X))$$

The simple vending machine has an alphabet of two symbols, namely, *coin* and *choc*. The behaviour of the machine is that a coin is entered into the machine, and then a chocolate selected and provided, and the machine is ready for further use.

Fig. 5.1 Deterministic finite
state machine
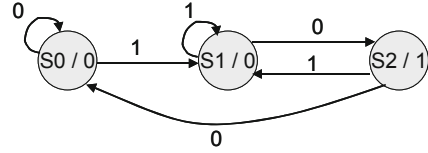
CSP processes use channels to communicate values with their environment, and
input on channel c is denoted by $(c?.x\ P_x)$. This describes a process that accepts any
value x on channel c, and then behaves as process P_x. In contrast, $(c!e\ P)$ defines a
process which outputs the expression e on channel c and then behaves as process P.

The π-calculus is a process calculus based on names. Communication between
processes takes place between known channels, and the name of a channel may be
passed over a channel. There is no distinction between channel names and data values
in the π-calculus. The output of a value v on channel a is given by $\bar{a}v$; i.e. output
is a negative prefix. Input on a channel a is given by $a(x)$, and is a positive prefix.
Private links or restrictions are given by $(x)P$ in the π-calculus.

5.14   Finite State Machines

The neurophysiologists Warren McCulloch and Walter Pitts published early work
on finite state automata in 1943. They were interested in modelling the thought
process for humans and machines. Moore and Mealy developed this work further,
and these finite-state machines are referred to as the "Mealy machine" and the "Moore
machine". The Mealy machine determines its outputs through the current state and
the input, whereas the output of Moore's machine is based upon the current state
alone.

Definition 5.2 (Finite State Machine) *A finite state machine (FSM) is an abstract
mathematical machine that consists of a finite number of states. It includes a start
state q_0 in which the machine is in initially; a finite set of states Q; an input alphabet
Σ; a state transition function δ; and a set of final accepting states F (where $F \subseteq Q$).*

The state transition function takes the current state and an input and returns the
next state. That is, the transition function is of the form:

$$\delta : Q \times \Sigma \to Q$$

The transition function provides rules that define the action of the machine for each
input, and it may be extended to provide output as well as a state transition. State
diagrams are used to represent finite state machines, and each state accepts a finite
number of inputs. A finite state machine may be deterministic or non-deterministic,
and a deterministic machine changes to exactly one state for each input transition,
whereas a non-deterministic machine may have a choice of states to move to for a
particular input (Fig. 5.1).

Finite state automata can compute only very primitive functions and are not an adequate model for computing. There are more powerful automata such as the Turing machine that is essentially a finite automaton with an infinite storage (memory). Anything that is computable is computable by a Turing machine.

The memory of the Turing machine is a tape that consists of an infinite number of one-dimensional cells. The Turing machine provides a mathematical abstraction of computer execution and storage, as well as providing a mathematical definition of an algorithm.

5.15   The Parnas Way

Parnas has been influential in the computing field, and his ideas on the specification, design, implementation, maintenance, and documentation of computer software remain important. He advocates a solid engineering approach and argues that the role of the engineer is to apply scientific principles and mathematics to design and develop products. He argues that computer scientists need to be educated as engineers to ensure that they have the appropriate background to build software correctly. His contributions to software engineering include:

- *Tabular Expressions*
  These are mathematical tables for specifying requirements and enable complex predicate logic expressions to be represented in a simpler form.
- *Mathematical Documentation*
  He advocates the use of precise mathematical documentation for requirements and design.
- *Requirements Specification*
  He advocates the use of mathematical relations to specify the requirements precisely.
- *Software Design*
  He developed *information hiding* that is used in object-oriented design,[10] and allows software to be designed for change. Every information-hiding module has an interface that provides the only means to access the services provided by the modules. The interface hides the module's implementation.
- *Software Inspections*
  His approach requires the reviewers to take an active part in the inspection. They are provided with a list of questions by the author and their analysis involves the production of mathematical table to justify the answers.
- *Predicate Logic*
  He developed an extension of the predicate calculus to deal with partial functions. This approach preserves the classical two-valued logic and deals with undefined values that may occur in predicate logic expressions.

---

[10] It is surprising that many in the object-oriented world seem unaware that information hiding goes back to the early 1970s and many have never heard of Parnas.

5.16   Usability of Formal Methods

There are practical difficulties associated with the use of formal methods. It seems to be assumed that programmers and customers are willing to become familiar with the mathematics used in formal methods. There is little evidence to suggest that customers would be prepared to use formal methods.[11] Customers are concerned with their own domain and speak the technical language of that domain.[12] Often, the use of mathematics is an alien activity that bears little resemblance to their normal work. Programmers are interested in programming rather than in mathematics, and generally are not interested in becoming mathematicians.[13]

However, the mathematics involved in most formal methods is reasonably elementary, and, in theory, if both customers and programmers are willing to learn the formal mathematical notation, then a rigorous validation of the formal specification can take place to verify its correctness. Both parties can review the formal specification to ensure its correctness, and the code can be verified to be correct with respect to the formal specification. It is usually possible to get a developer to learn a formal method, as a programmer has some experience of mathematics and logic; however, in practice, it is more difficult to get a customer to learn a formal method.

This often means that a formal specification of the requirements and an informal definition of the requirements using a natural language are maintained. It is essential that both of these documents are consistent and that there is a rigorous validation of the formal specification. Otherwise, if the programmer proves the correctness of the code with respect to the formal specification, and the formal specification is incorrect, then the formal development of the software is incorrect. There are several techniques to validate a formal specification and these are described in [Wic:00] (also see Table 5.2).

5.16.1   Why Are Formal Methods Difficult?

Formal methods are perceived as being difficult to use and of providing limited value in mainstream software engineering. Programmers receive some training in mathematics as part of their education. However, in practice, most programmers who learn formal methods at university never use formal methods again once they take an industrial position.

---

[11] The domain in which the software is being used will influence the willingness or otherwise of the customers to become familiar with the mathematics required. Certainly, in mainstream software engineering the author does not detect any interest from customers and the perception is that formal methods are unusable; however, in some domains such as the regulated sector there is a greater willingness of customers to become familiar with the mathematical notation.

[12] The author's experience is that most customers have a very limited interest and even less willingness to use mathematics. There are exceptions to this especially in the regulated sector.

[13] Mathematics that is potentially useful to software engineers is discussed in Chap. 2.

Table 5.2 Techniques for validation of formal specification

Technique	Description
Proof	This involves demonstrating that the formal specification satisfies key properties of the requirements. The implementation will need to preserve these properties
Software inspections	This involves a Fagan-like inspection to compare an informal set of requirements (unless the customer has learned the formal method) with the formal specification, and to ensure consistency between them
Specification animation	This involves program (or specification) execution as a way to validate the formal specification. It is similar to testing
Tools	Tools provide some limited support in validating a formal specification

Table 5.3 Factors in difficulty of formal methods

Factor	Description
Notation/intuition	The notation employed differs from that employed in mathematics. Intuition varies from person to person. Many programmers find the notation in formal methods to be unintuitive
Formal specification	It is easier to read a formal specification than to write one
Validation of formal specification	The validation of a formal specification using proof techniques or a Fagan-like inspection is difficult
Refinement[a]	The refinement of a formal specification into successive more concrete specifications with proof of validity of each refinement step is difficult and time consuming
Proof	Proof can be difficult and time consuming
Tool support	Many of the existing tools are difficult to use

[a]The author doubts that refinement is cost effective for mainstream software engineering. However, it may be useful in the regulated environment

It may well be that the very nature of formal methods is such that it is suited only for specialists with a strong background in mathematics. Some of the reasons why formal methods are perceived as being difficult are discussed in Table 5.3.

5.16.2   *Characteristics of a Usable Formal Method*

It is important to investigate ways by which formal methods can be made more usable to software engineers. This may involve designing more usable notations and better tools to support the process. Practical training and coaching to employees can help also. Some of the characteristics of a usable formal method are discussed in Table 5.4.

Table 5.4 Characteristics of a usable formal method

Characteristic	Description
Intuitive	A formal method should be intuitive
Teachable	A formal method needs to be teachable to the average software engineer. The training should include (at least) writing practical formal specifications
Tool support	Good tools to support formal specification, validation, refinement and proof are required
Adaptable to change	Change is common in a software engineering environment. A usable formal method should be adaptable to change
Technology transfer path	The process for software development needs to be defined to include formal methods. The migration to formal methods needs to be managed
Cost	The use of formal methods should be cost effective with a return on investment. There should be benefits in time, quality and productivity

[a]A commercial company will expect a return on investment from the use of a new technology. This may be reduced software development costs, improved quality, improved timeliness of projects or improvements in productivity. A company does not go to the trouble of deploying a new technology just to satisfy academic interest

5.17   Review Questions

1. What are formal methods and describe their potential benefits? How essential is tool support?
2. What is stepwise refinement and is it realistic in mainstream software engineering?
3. Discuss Parnas's criticisms of formal methods and discuss whether his views are valid.
4. Discuss the applications of formal methods and which areas have benefited most from their use? What problems have arisen?
5. Describe a technology transfer path for the potential deployment of formal methods in an organisation.
6. Explain the difference between the model-oriented approach and the axiomatic approach.
7. Discuss the nature of proof in formal methods and tools to support proof.
8. Discuss the Vienna Development Method (VDM) and explain the difference between standard VDM and VDM♣.
9. Discuss Z and B? Describe the tools in the B-Toolkit.
10. Discuss process calculi such as CSP, CCS or π-calculus.

5.18   Summary

This chapter discussed formal methods, which is a rigorous approach to the development of high-quality software. Formal methods employ mathematical techniques for the specification and formal development of software, and are very useful in the

safety-critical field. They consist of formal specification languages or notations; a methodology for formal software development; and a set of tools to support the syntax checking of the specification, as well as the proof of properties of the specification.

The use of formal methods generally leads to more robust software and to increased confidence in its correctness. There are challenges involved in the deployment of formal methods, as the use of these mathematical techniques may be a culture shock to many staff.

Formal methods allow questions to be asked and answered about what the system does independently of the implementation.

Formal methods may be model oriented or axiomatic oriented. The model-oriented approach includes formal methods such as VDM, Z and B. The axiomatic approach includes the process calculi such as CSP, CCS and the π calculus.

The usability of existing formal methods was considered. The reasons for the difficulty with formal methods was investigate, and the characteristics of a usable formal method explored.

Chapter 6
Z Formal Specification Language

Key Topics
    Sets, relations and functions
    Bags and sequences
    Data Reification and Refinement
    Schema Calculus
    Proof in Z

6.1 Introduction

Z is a formal specification language based on Zermelo set theory. It was developed at the Programming Research Group at Oxford University in the early 1980s [Dil:90], and became an ISO standard in 2002. Z specifications are mathematical and employ a classical two-valued logic. The use of mathematics ensures precision and allows inconsistencies and gaps in the specification to be identified. Theorem provers may be employed to demonstrate that the software implementation meets its specification.

Z is a 'model oriented' approach with an explicit model of the state of an abstract machine given, and operations are defined in terms of this state. Its mathematical notation is used for formal specification, and the schema calculus is used to structure the specifications. It is visually striking and consists essentially of boxes, with these boxes or schemas used to describe operations and states. The schema calculus enables schemas to be used as building blocks and combined with other schemas. The simple schema shown in Fig. 6.1 is the specification of the positive square root of a real number.

The schema calculus is a powerful means of decomposing a specification into smaller pieces or schemas. This helps to make Z specifications highly readable, as each individual schema is small in size and self-contained. Exception handling is addressed by defining schemas for the exception cases. These are then combined with the original operation schema. Mathematical data types are used to model the data in a system and these data types obey mathematical laws. These laws enable simplification of expressions and are useful with proofs.

Fig. 6.1 Specification of
positive square root

$$
\begin{array}{|l}
\hline
\,SqRoot \underline{\hspace{2cm}} \\
\;num?,\ root!\ :\ \mathbb{R} \\
\underline{\hspace{1cm}} \\
num? \geq 0 \\
root!^2\ =\ num? \\
root!\ \geq 0 \\
\underline{\hspace{2cm}}
\end{array}
$$

Operations are defined in a pre-condition/post-condition style. A pre-condition must be true before the operation is executed, and the post-condition must be true after the operation has executed. The pre-condition is implicitly defined within the operation. Each operation has an associated proof obligation to ensure that if the pre-condition is true, then the operation preserves the system invariant. The system invariant is a property of the system that must be true at all times. The initial state itself is, of course, required to satisfy the system invariant.

The pre-condition for the specification of the square root function above is that $num? \geq 0$; i.e. the function *SqRoot* may be applied to positive real numbers only. The post-condition for the square root function is $root!^2 = num?$ and $root! \geq 0$. That is, the square root of the number is positive and its square gives the number. Post-conditions employ a logical predicate which relates the pre-state to the post-state, and the post-state of a variable being distinguished by priming the variable, e.g. v'.

Z is a typed language and whenever a variable is introduced, its type must be given. A type is simply a collection of objects, and there are several standard types in Z. These include the natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$ and the real numbers $\mathbb{R}$. The declaration of a variable x of type X is written $x : X$. It is also possible to create your own types in Z.

Various conventions are employed within Z specification, for example $v?$ indicates that v is an input variable; $v!$ indicates that v is an output variable. The variable *num?* is an input variable and *root!* is an output variable for the square root example shown in Fig. 6.1. The notation Ξ in a schema indicates that the operation *Op* does not affect the state; whereas the notation Δ in the schema indicates that *Op* is an operation that affects the state.

Many of the data types employed in Z have no counterpart in standard programming languages. It is therefore important to identify and describe the concrete data structures that ultimately will represent the abstract mathematical structures. As the concrete structures may differ from the abstract, the operations on the abstract data structures may need to be refined to yield operations on the concrete data that yield equivalent results. For simple systems, direct refinement (i.e. one step from abstract specification to implementation) may be possible; in more complex systems, deferred refinement[1] is employed, where a sequence of increasingly concrete specifications

---

[1] Step-wise refinement involves producing a sequence of increasingly more concrete specifications until eventually the executable code is produced. Each refinement step has associated proof obligations to prove that the refinement step is valid.

Fig. 6.2 Specification of a
library system

```
–Library————
on-shelf, missing, borrowed : ℙ Bkd-Id
————
on-shelf ∩ missing = Ø
on-shelf ∩ borrowed = Ø
borrowed ∩ missing = Ø
```

Fig. 6.3 Specification of
borrow operation

```
–Borrow————
Λ Library
b? :Bkd-Id
————
b? ∈ on-shelf
on-shelf' = on-shelf \ {b?}
borrowed' = borrowed ∪ {b?}
```

are produced to yield the executable specification. There is a calculus for combining schemas to make larger specifications, and this is discussed later in this chapter.

Example 6.1 The following is a Z specification to borrow a book from a library system. The library is made up of books that are on the shelf; books that are borrowed and books that are missing. The specification models a library with sets representing books on the shelf, on loan or missing. These are three mutually disjoint subsets of the set of books *Bkd-Id*.

The system state is defined in the *Library* schema in Fig. 6.2 and operations such as *Borrow* and *Return* affect the state. The *Borrow* operation is specified in Fig. 6.3.

The notation ℙ*Bkd-Id* is used to represent the power set of *Bkd-Id* (i.e. the set of all subsets of *Bkd-Id*). The disjointness condition for the library is expressed by the requirement that the pair-wise intersection of the subsets *on-shelf*, *borrowed*, *missing* is the empty set (shown in Fig. 6.3).

The pre-condition for the *Borrow* operation is that the book must be available on the shelf to borrow. The post-condition is that the borrowed book is added to the set of borrowed books and is removed from the books on the shelf.

Z has been successfully applied in industry including the CICS project at IBM Hursley in the United Kingdom.[2] Next, we describe the key parts of Z including sets, relations, functions, sequences and bags.

6.2 Sets

Sets were discussed in Chap. 2 and this section focuses on their use in Z. Sets may be enumerated by listing all of their elements. Thus, the set of all even natural numbers less than or equal to 10 is:

$$\{2, 4, 6, 8, 10\}.$$

---

[2] This project claimed a 9 % increase in productivity attributed to the use of formal methods.

Sets can be created from other sets using set comprehension: i.e. stating the properties that its members must satisfy. For example, the set of even natural numbers less than 10 is given by set comprehension as:

$$\{n : \mathbb{N} | n \neq 0 \wedge n < 10 \wedge n \bmod 2 = 0 \bullet n.$$

There are three main parts to the set comprehension above. The first part is the signature of the set and this is given by $n : \mathbb{N}$ above. The first part is separated from the second part by a vertical line. The second part is given by a predicate, and for this example the predicate is $n \neq 0 \wedge n < 10 \wedge n \bmod 2 = 0$. The second part is separated from the third part by a bullet. The third part is a term, and for this example it is simply n. The term is often a more complex expression: e.g. $\log(n^2)$.

In mathematics, there is just one empty set. However, since Z is a typed set theory, there is an empty set for each type of set. Hence, there are an infinite number of empty sets in Z. The empty set is written as $\emptyset[X]$ where X is the type of the empty set. In practice, X is omitted when the type is clear.

Various set operations such as union, intersection, set difference and symmetric difference are employed in Z. The powerset of a set X is the set of all subsets of X and is denoted by $\mathbb{P}X$. The set of non-empty subsets of X is denoted by $\mathbb{P}_1 X$ where

$$\mathbb{P}_1 X == \{U : \mathbb{P}X | U \neq \emptyset[X]\}.$$

A finite set of elements of type X (denoted by $F\ X$) is a subset of X that cannot be put into a one-to-one correspondence with a proper subset of itself. This is defined formally as:

$$F X == \{U : \mathbb{P}X | \neg \exists V : \mathbb{P}U \bullet V \neq U \wedge (\exists f : V > \rightarrow U\}.$$

The expression $f : V > \rightarrow U$ denotes that f is a bijection from U to V and injective, surjective and bijective functions were discussed in Chap. 2.

The fact that Z is a typed language means that whenever a variable is introduced (e.g. in quantification with $\forall$ and $\exists$) it is first declared. For example, $\forall j{:}J \bullet P \Rightarrow Q$. There is also the unique existential quantifier $\exists_1 j{:}J | P$ which states that there is exactly one j of type J that has property P.

6.3   Relations

Relations are used extensively in Z and were discussed in Chap. 2. A relation R between X and Y is any subset of the Cartesian product of X and Y; i.e. $R \subseteq (X \times Y)$, and the relation is denoted by $R : X \leftrightarrow Y$. The notation $x \mapsto y$ indicates that the pair $(x, y) \in R$.

Consider, the relation *home_owner* : *Person* $\leftrightarrow$ *Home* that exists between people and their homes. An entry *daphne* $\mapsto$ *mandalay* $\in$ *home_owner* if *daphne* is the

owner of *mandalay*. It is possible for a person to own more than one home:

$$rebecca \mapsto nirvana \in home\_owner$$

$$rebecca \mapsto tivoli \in home\_owner.$$

It is possible for two people to share the ownership of a home:

$$rebecca \mapsto nirvana \in home\_owner$$

$$lawrence \mapsto nirvana \in home\_owner.$$

There may be some people who do not own a home, and there is no entry for these people in the relation *home_owner*. The type *Person* includes every possible person, and the type *Home* includes every possible home. The domain of the relation *home_owner* is given by:

$$x \in \text{dom } home\_owner \Leftrightarrow \exists h : Home \bullet x \mapsto h \in home\_owner.$$

The range of the relation *home_owner* is given by:

$$h \in \text{ran } home\_owner \Leftrightarrow \exists x : Person \bullet x \mapsto h \in home\_owner.$$

The composition of two relations *home_owner* : *Person* $\leftrightarrow$ *Home* and *home_value* : *Home* $\leftrightarrow$ *Value* yields the relation *owner_wealth* : *Person* $\leftrightarrow$ *Value* and is given by the relational composition *home_owner*; *home_value* where:

$$p \mapsto v \in home\_owner \,;\, home\_value \Leftrightarrow$$

$$(\exists h : Home \bullet p \mapsto h \in home\_owner \wedge h \mapsto v \in home\_value).$$

The relational composition may also be expressed as:

$$owner\_wealth = home\_value \circ home\_owner.$$

The union of two relations often arises in practice. Suppose a new entry *aisling* $\mapsto$ *muckross* is to be added. Then this is given by

$$home\_owner' = home\_owner \cup \{aisling \mapsto muckross\}$$

Suppose that we are interested in knowing all females who are house owners. Then, we restrict the relation *home_owner* so that the first element of all ordered pairs have to be female. Consider *female* : $\mathbb{P}$ *Person* with $\{aisling, rebecca\} \subseteq female$.

$$home\_owner = \{aisling \mapsto muckross, rebecca \mapsto nirvana, lawrence \mapsto nirvana\}$$

$$female \triangleleft home\_owner = \{aisling \mapsto muckross, rebecca \mapsto nirvana\}.$$

That is, *female* $\triangleleft$ *home_owner* is a relation that is a subset of *home_owner*, and the first element of each ordered pair in the relation is female. The operation $\triangleleft$ is termed domain restriction and its fundamental property is:

$$x \mapsto y \in U \triangleleft R \Leftrightarrow (x \in U \wedge x \mapsto y \in R).$$

Where $R : X \leftrightarrow Y$ and $U : \mathbb{P} X$.

There is also a domain anti-restriction (subtraction) operation and its fundamental property is:

$$x \mapsto y \in U \lhd R \Leftrightarrow (x \notin U \wedge x \mapsto y \in R)$$

Where $R : X \leftrightarrow Y$ and $U : \mathbb{P}X$.

There are also range restriction (the $\rhd$ operator) and the range anti-restriction operator (the $\rhd$ operator). These are discussed in [Dil:90].

6.4   Functions

A function is an association between objects of some type X and objects of another type Y such that given an object of type X, there exists only one object in Y associated with that object [Dil:90]. That is, a function is a set of ordered pairs where the first element of the ordered pair has at most one element associated with it. Therefore, a function is a special type of relation, and it can be *total* or *partial*.

A total function has exactly one element in Y associated with each element of X, whereas a partial function has at most one element of Y associated with each element of X (there may be elements of X that have no element of Y associated with them).

A partial function from X to Y (denoted $f : X \nrightarrow Y$) is a relation $f : X \leftrightarrow Y$ such that:

$$\forall x : X \,; y, z : Y \bullet (x \mapsto y \in f \wedge x \rightarrow z \in f \Rightarrow y = z).$$

The association between x and y is denoted by $f(x) = y$, and this indicates that the value of the partial function f at x is y. A total function from X to Y (denoted $f : X \rightarrow Y$) is a partial function such that every element in X is associated with some value of Y.

$$f : X \rightarrow Y \Leftrightarrow f : X \nrightarrow Y \wedge \operatorname{dom} f = X.$$

Clearly, every total function is a partial function but not vice versa.

One operation that arises quite frequently in specifications is the function override operation. Consider the following specification of a temperature map:

$$
\begin{array}{|l}
\hline
\text{--}TempMap\text{---------} \\
CityList : \mathbb{P}\,City \\
temp : City \nrightarrow \mathbb{Z} \\
\hline
\operatorname{dom} temp = CityList \\
\hline
\end{array}
$$

Suppose the temperature map is given by $temp = \{Cork \mapsto 17, Dublin \mapsto 19, London \mapsto 15\}$. Then consider the problem of updating the temperature map, if a new temperature reading is made in Cork say $\{Cork \mapsto 18\}$. Then, the new temperature chart is obtained from the old temperature chart by function override to yield $\{Cork \mapsto 18, Dublin \mapsto 19, London \mapsto 15\}$. This is written as:

$$temp' = temp \oplus \{Cork \mapsto 18\}.$$

The function override operation combines two functions of the same type to give a new function of the same type. The effect of the override operation is that the entry $\{Cork \mapsto 17\}$ is removed from the temperature chart and replaced with the entry $\{Cork \mapsto 18\}$.

Suppose $f, g : X + \mapsto Y$ are partial functions, then $f \oplus g$ is defined and indicates that f is overridden by g. It is defined as follows:

$$(f \oplus g)(x) = \begin{cases} g(x) & \text{where } x \in \text{dom } g \\ f(x) & \text{where } x \notin \text{dom } g \wedge x \in \text{dom } f \end{cases}.$$

This may also be expressed (using function override) as:

$$(f \oplus g) = ((\text{dom } g) \lhd f) \cup g.$$

There is notation in Z for injective, surjective and bijective functions. An injective function is one-to-one: i.e.

$$f(x) = f(y) \Rightarrow x = y.$$

A surjective function is onto: i.e.

$$\text{Given } y \in Y, \exists x \in X \text{ such that } f(x) = y.$$

A bijective function is one-to-one and onto, and it indicates that the sets X and Y can be put into one-to-one correspondence with one another. Z includes lambda calculus notation (λ-notation) to define functions. For example, the function cube $== \lambda x : N \bullet x \times x \times x$. Lambda calculus is discussed in Chap. 10. Function composition $f ; g$ is similar to relational composition.

6.5   Sequences

The type of all sequences of elements drawn from a set X is denoted by seq X. Sequences are written as $\langle x_1, x_2, \ldots, x_n \rangle$ and the empty sequence is denoted by $\langle \rangle$. Sequences may be used to specify the changing state of a variable over time with each element of the sequence representing the value of the variable at a discrete time instance.

Sequences are functions and a sequence of elements drawn from a set X is a finite function from the set of natural numbers to X. A partial finite function f from X to Y is denoted by $f : X + \mapsto Y$. A finite sequence of elements of X is given by $f : N + \mapsto X$, and the domain of the function consists of all numbers between 1 and $\#f$. It is defined formally as:

$$\text{seq } X == \{f : N + \mapsto X \mid \text{dom } f = 1 \ldots \#f \bullet f\}.$$

The sequence $\langle x_1, x_2, \ldots x_n \rangle$ above is given by:

$$\{1 \mapsto x_1, 2 \mapsto x_2, \ldots, n \mapsto x_n\}.$$

There are various functions to manipulate sequences. These include the sequence concatenation operation. Suppose $\sigma = \langle x_1, x_2, \ldots, x_n \rangle$ and $\tau = \langle y_1, y_2, \ldots, y_m \rangle$, then:

$$\sigma^\frown \tau = \langle x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \rangle.$$

The head of a non-empty sequence gives the first element of the sequence:

$$\text{head } \sigma = \text{head } \langle x_1, x_2, \ldots, x_n \rangle = x_1.$$

The tail of a non-empty sequence is the same sequence except that the first element of the sequence is removed:

$$\text{tail } \sigma = \text{tail } \langle x_1, x_2, \ldots, x_n \rangle = \langle x_2, \ldots, x_n \rangle.$$

Suppose $f : X \to Y$ and a sequence $\sigma : \text{seq } X$, then the function map applies f to each element of σ:

$$\text{map } f \sigma = \text{map } f \langle x_1, x_2, \ldots, x_n \rangle = \langle f(x_1), f(x_2), \ldots, f(x_n) \rangle.$$

The map function may also be expressed via function composition as:

$$\text{map } f \sigma = \sigma ; f.$$

The reverse order of a sequence is given by the rev function:

$$\text{rev} \sigma = \text{rev} \langle x_1, x_2, \ldots, x_n \rangle = \langle x_n, \ldots, x_2, x_1 \rangle.$$

6.6   Bags

A bag is similar to a set except that there may be multiple occurrences of each element in the bag. A bag of elements of type X is defined as a partial function from the type of the elements of the bag to positive whole numbers. The definition of a bag of type X is:

$$\text{bag } X == X \nrightarrow \mathbb{N}_1.$$

For example, a bag of marbles may contain 3 blue marbles, 2 red marbles and 1 green marble. This is denoted by $B = [|b, b, b, g,, r, r]$. The bag of marbles is thus denoted by:

$$\text{bag } Marble == Marble \nrightarrow \mathbb{N}_1.$$

Fig. 6.4 Specification of
vending machine using bags

$$\begin{array}{|l}\hline\!\!-\Delta\textit{Vending Machine}\!\!-\!\!-\!\!-\\ \textit{stock}:\text{bag }\textit{Good}\\ \textit{price}:\textit{Good}\rightarrow\mathbb{N}_1\\ \hline \text{dom }\textit{stock}\subseteq\text{dom }\textit{price}\\ \hline\end{array}$$

The function count determines the number of occurrences of an element in a bag. For the example above, count *Marble b* = 3, and count *Marble y* = 0 since there are no yellow marbles in the bag. This is defined formally as:

$$\text{count bag } X \ y = \begin{cases} 0 & y \notin \text{bag } X \\ (\text{bag } X)(y) & y \in \text{bag } X \end{cases}.$$

An element y is in bag X if and only if y is in the domain of bag X.

$$y \text{ in bag } X \Leftrightarrow y \in \text{dom (bag } X).$$

The union of two bags of marbles $B_1 = [|b,b,b,g,,r,r|]$ and $B_2 = [|b,,g,,r,y|]$ is given by $B_1 \uplus B_2 = [|b,b,b,b,g,g,r,r,r,y|]$. It is defined formally as:

$$(B_1 \uplus B_2)(y) = \begin{cases} B_2(y) & y \notin \text{dom } B_1 \wedge y \in \text{dom } B_2 \\ B_1(y) & y \in \text{dom } B_1 \wedge y \notin \text{dom } B_2 \\ B_1(y) + B_1(y) & y \in \text{dom } B_1 \wedge y \in \text{dom } B_2 \end{cases}.$$

A bag may be used to record the number of occurrences of each product in a warehouse as part of an inventory system. It may model the number of items remaining for each product in a vending machine (Fig. 6.4).

The operation of a vending machine would require other operations such as identifying the set of acceptable coins, checking that the customer has entered sufficient coins to cover the cost of the good, returning change to the customer and updating the quantity on hand of each good after a purchase. A more detailed examination is in [Dil:90].

6.7   Schemas and Schema Composition

The schemas in Z are visually striking and the specification is presented in two-dimensional graphic boxes. Schemas are used for specifying states and state transitions, and they employ notation to represent the before and after state (e.g. s and s' where s' represents the after state of s). They group all relevant information that belongs to a state description.

There are a number of useful schema operations such as schema inclusion, schema composition and the use of propositional connectives to link schemas together. The

Fig. 6.5 Schema inclusion

$$
\begin{array}{|l}
\hline \text{–}S_2\text{———} \\
x,y : \mathbb{N} \\
z : \mathbb{N} \\
\hline
x + y > 2 \\
z = x + y \\
\hline
\end{array}
$$

Fig. 6.6 Merging schemas
$(S_1 \vee S_2)$

$$
\begin{array}{|l}
\hline \text{–}S\text{———} \\
x,y : \mathbb{N} \\
z : \mathbb{N} \\
\hline
x + y > 2 \vee z = x + y \\
\hline
\end{array}
$$

Δ convention indicates that the operation affects the state whereas the Ξ convention indicates that the state is not affected. These operations and conventions allow complex operations to be specified concisely, and assist with the readability of the specification. Schema composition is analogous to relational composition and allows new schemas to be derived.

A schema name S_1 may be included in the declaration part of another schema S_2. The effect of the inclusion is that the declarations in S_1 are now part of S_2 and the predicates of S_1 are S_2 are joined together by conjunction. If the same variable is defined in both S_1 and S_2, then it must be of the same type in both schemas.

$$
\begin{array}{|l}
\hline \text{–}S_1\text{———} \\
x,y : \mathbb{N} \\
\hline
x + y > 2 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \text{–}S_2\text{———} \\
S_1 ; z : \mathbb{N} \\
\hline
z = x + y \\
\hline
\end{array}
$$

The result is that S_2 includes the declarations and predicates of S_1 (Fig. 6.5).

Two schemas may be linked by propositional connectives such as $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$, and $S_1 \leftrightarrow S_2$. The schema $S_1 \vee S_2$ is formed by merging the declaration parts of S_1 and S_2, and then combining their predicates with the logical $\vee$ operator. For example, $S = S_1 \vee S_2$ yields (Fig. 6.6):

Schema inclusion and the linking of schemas use normalisation to convert sub-types to maximal types, and predicates are employed to restrict the maximal type to the sub-type. This involves replacing declarations of variables (e.g. $u : 1, \ldots, 35$ with $u : Z$, and adding the predicate $u > 0$ and $u < 36$ to the predicate part of the schema).

The Δ and Ξ conventions are used extensively, and the notation Δ *TempMap* is used in the specification of schemas that involve a change of state. The notation Δ *TempMap* represents:

$$\Delta TempMap = TempMap \wedge TempMap'.$$

Table 6.1 Schema composition

Step	Procedure
1	Rename all *after*-state variables in S to something new: $S\,[s^+/s']$
2	Rename all *before* state variables in T to the same new thing: i.e. $T\,[s^+/s]$
3	Form the conjunction of the two new schemas: $S\,[s^+/s'] \wedge T\,[s^+/s]$
4	Hide the variable introduced in step 1 and 2 $S\,;T = (S\,[s^+/s'] \wedge T\,[s^+/s])/(s^+)$

The longer form of Δ *TempMap* is written as:

$$
\begin{array}{|l}
\!-\!\Delta TempMap\!-\!\!-\!\!-\!\!- \\
CityList,\ CityList' : \mathbb{P}\ \ City \\
temp,\ temp' : City + \mapsto \mathbb{Z} \\
\hline
\text{dom}\ temp = CityList \\
\text{dom}\ temp' = CityList' \\
\hline
\end{array}
$$

The notation Ξ *TempMap* is used in the specification of operations that do not involve a change to the state. It represents:

$$
\begin{array}{|l}
\!-\!\Xi\ TempMap\!-\!\!-\!\!-\!\!- \\
\Delta TempMap \\
\hline
CityList = CityList' \\
temp = temp' \\
\hline
\end{array}
$$

Schema composition is analogous to relational composition and it allows new specifications to be built from existing ones. It allows the after-state variables of one schema to be related with the before variables of another schema. The composition of two schemas S and T ($S\,;T$) is described in detail in [Dil:90] and involves four steps (Table 6.1):

The example below should make schema composition clearer. Consider the composition of S and T where S and T are defined as follows:

$$
\begin{array}{|l}
\!-\!S\!-\!\!-\!\!- \\
x,x',y? : \mathbb{N} \\
\hline
x' = y? \text{ - } 2 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\!-\!T\!-\!\!-\!\!- \\
x,x' : \mathbb{N} \\
\hline
x' = x + 1 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\!-\!S_1\!-\!\!-\!\!- \\
x,x^+,y? : \mathbb{N} \\
\hline
x^+ = y? \text{ - } 2 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\!-\!T_1\!-\!\!-\!\!- \\
x^+,x' : \mathbb{N} \\
\hline
x' = x^+ + 1 \\
\hline
\end{array}
$$

S_1 and T_1 represent the results of step 1 and 2, with x' renamed to x^+ in S, and x renamed to x^+ in T. Step 3 and 4 yield Fig. 6.7.

Schema composition allows new specifications to be created from existing ones.

Fig. 6.7 Schema composition

$$
\begin{array}{|l}
\hline S_1 \wedge T_1 \quad\quad \\
x, x^+, x', y? : \mathbb{N} \\
\hline
x^+ = y? - 2 \\
x' = x^+ + 1 \\
\hline
\end{array}
\quad\quad
\begin{array}{|l}
\hline S \; ; \; T \quad\quad \\
x, \; x', \; y? \; : \mathbb{N} \\
\hline
\exists x^+ : \mathbb{N} \; \bullet \\
\quad (x^+ = y? - 2 \\
\quad\quad x' = x^+ + 1) \\
\hline
\end{array}
$$

6.8   Reification and Decomposition

A Z specification involves defining the state of the system and then specifying the required operations. The Z specification language employs many constructs that are not part of conventional programming languages, and it is therefore not directly executable on a computer. A programmer implements the formal specification, and mathematical proof may be employed to prove that a program meets its specification.

Often, there is a need to write an intermediate specification that is between the original Z specification and the eventual program code. This intermediate specification is more algorithmic and uses less abstract data types than the Z specification. It is termed the design and needs to be correct with respect to the specification, and the program needs to be correct with respect to it. The design is a refinement (reification) of the specification, and the operations of the specification have been decomposed into those of the design.

The representation of an abstract data type such as a set by a sequence is termed data reification, and this is concerned with the process of transforming an abstract data type into a concrete data type. The abstract and concrete data types are related by the retrieve function, which maps the concrete data type to the abstract data type. There are typically several possible concrete data types for a particular abstract data type (i.e. refinement is a relation), whereas there is one abstract data type for a concrete data type (i.e. retrieval is a function). For example, sets are often reified to unique sequences; however, more than one unique sequence can represent a set whereas a unique sequence represents exactly one set.

The operations defined on the concrete data type are related to the operations defined on the abstract data type. That is, the commuting diagram property is required to hold; i.e. the operation $\boxdot$ on the concrete data type correctly refines the operation $\odot$ on the abstract data type if the following diagram commutes, and this property commuting diagram property requires proof (Fig 6.8).

That is, it is required to prove that:

$$ret(\sigma \boxdot \tau) = (ret\,\sigma) \odot (ret\,\tau).$$

In Z, the refinement and decomposition is done with schemas. It is required to prove that the concrete schema is a valid refinement of the abstract schema, and this gives rise to a number of proof obligations. It needs to be proved that the initial states

Fig. 6.8 Refinement
commuting diagram

$$retr(\sigma) \odot retr(\tau)$$

$retr(\sigma),$
$retr(\tau)$

$retr$
$(\sigma \boxdot \tau)$

$$(\sigma \boxdot \tau)$$
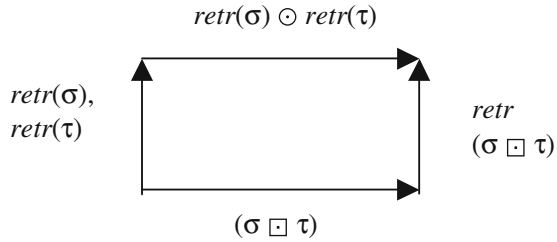
correspond to one another, and that each operation in the concrete schema is correct with respect to the operation in the abstract schema, and also that it is applicable (i.e. whenever the abstract operation may be performed the concrete operation may also be performed).

6.9   Proof in Z

Mathematicians perform rigorous proof of theorems using technical and natural language. Logicians employ formal proofs to prove theorems using propositional and predicate calculus. Formal proofs generally involve a long chain of reasoning with every step of the proof justified. Rigorous proofs involve precise reasoning using a mixture of natural and mathematical language. Rigorous proofs [Dil:90] have been described as being analogous to high-level programming languages, whereas formal proofs are analogous to machine language.

A mathematical proof includes natural language and mathematical symbols and often many of the tedious details of the proof are omitted. Many proofs in formal methods such as Z are concerned with crosschecking on the details of the specification, or on the validity of the refinement step, or proofs that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and tool support is essential as proof by hand often contain errors or jumps in reasoning. Machine proofs are lengthy and largely unreadable; however, they provide extra confidence as every step in the proof is justified.

A formal mathematical proof consists of a sequence of formulae, where each element is either an axiom or derived from a previous element in the series by applying a fixed set of mechanical rules. The proof of various properties about the programmes increases confidence in its correctness.

6.10   Review Questions

1. Describe the main features of the Z specification language.
2. Explain the difference between $\mathbb{P}_1 X$, $\mathbb{P} X$ and $\pmb{F} X$.

3. Give an example of a set derived from another set using set comprehension. Explain the three main parts of set comprehension in Z.
4. Discuss the applications of Z and which areas have benefited most from their use? What problems have arisen?
5. Give examples to illustrate the use of domain and range restriction operators and domain and range anti-restriction operators with relations in Z.
6. Give examples to illustrate relational composition.
7. Explain the difference between a partial and total function and give examples to illustrate function override.
8. Give examples to illustrate the various operations on sequences including concatenation, head, tail, map and reverse operations.
9. Give examples to illustrate the various operations on bags.
10. Discuss the nature of proof in Z and tools to support proof.
11. Explain the process of refining an abstract schema to a more concrete representation, the proof obligations that are generated and the commuting diagram property.

6.11   Summary

Z was developed at Oxford University and it has been employed in both industry and academia. Its specifications are mathematical and this allows properties to be proved about the specification, and any gaps or inconsistencies in the specification may be identified.

Z is a model-oriented' approach and an explicit model of the state of an abstract machine is given, and the operations are defined in terms of their effect on the state. Its main features include a mathematical notation that is similar to Vienna Development Method (VDM) and the schema calculus. The latter consists essentially of boxes and is used to describe operations and states.

The schema calculus enables schemas to be used as building blocks to form larger specifications. It is a powerful means of decomposing a specification into smaller pieces and helps with the readability of Z specifications, as each individual schema is small in size and self-contained.

Z is a highly expressive specification language and includes notation for sets, functions, relations, bags, sequences, predicate calculus and schema calculus.

A Z specification may be refined into the detailed code. This involves producing intermediate specifications between the Z specification and the eventual program code. Mathematical proof is required to demonstrate the validity of the refinement step, and this involves a proof of the commuting diagram property.

Tool support is essential in conducting proof as hand proofs often involve jumps in reasoning and due to the volume of proof obligations.

Chapter 7
Number Theory

Key Topics
Square, Rectangular and Triangular Numbers
Prime Numbers
Pythagorean Triples
Mersenne Primes
Division Algorithm
Perfect and Amicable Numbers
Greatest Common Divisor
Least Common Multiples
Euclid's Algorithm
Modular Arithmetic

7.1   Introduction

Number theory is the branch of mathematics that is concerned with the mathematical properties of the natural numbers and integers. These include properties such as the parity of a number, divisibility, additive and multiplicative properties, whether a number is prime or composite, the prime factors of a number, the greatest common divisor and least common multiple of two numbers and so on.

Number theory has many applications in computing including cryptography and coding theory. For example, the RSA public key cryptographic system relies on its security due to the infeasibility of the integer factorisation problem for large numbers.

There are several unsolved problems in number theory and especially in prime number theory. For example, Goldbach's[1] conjecture states that every even integer greater than 2 is the sum of two primes, and this result has not been proved to date.

---

[1] Goldbach was an eighteenth-century German mathematician and Goldbach's conjecture has been verified to be true for all integers $n < 12 \times 10^{17}$.

Fig. 7.1 Pierre de Fermat

Fig. 7.2 Pythagorean triples
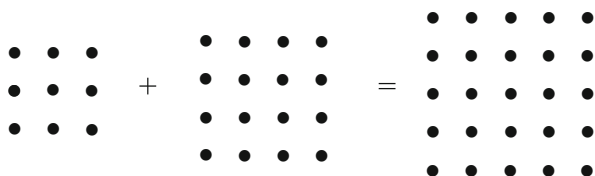
Fermat's[2] last theorem states that there is no integer solution to $x^n + y^n = z^n$ for $n > 2$, and this result remained unproved for over 300 years until Andrew Wiles finally proved it in the mid-1990s (Fig. 7.1).

The natural numbers $\mathbb{N}$ consist of the numbers $\{1, 2, 3, \ldots\}$. The integer numbers $\mathbb{Z}$ consist of $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The rational numbers $\mathbb{Q}$ consist of all numbers of the form $\{p/q$ where p and q are integers and $q \neq 0\}$. The real numbers $\mathbb{R}$ is defined to be the set of converging sequences of rational numbers and they are a superset of the rational numbers. They contain the rational and irrational numbers. The complex numbers $\mathbb{C}$ consist of all numbers of the form $\{a + bi$ where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}\}$.

Pythagorean triples are combinations of three whole numbers that satisfy Pythagoras's equation $x^2 + y^2 = z^2$. There are an infinite number of such triples, and an example of such a triple are the numbers 3, 4, 5 since $3^2 + 4^2 = 5^2$ (Fig. 7.2).

---

[2] Pierre de Fermat was a seventeenth-century French civil servant and amateur mathematician. He occasionally wrote to contemporary mathematicians announcing his latest theorem without providing the accompanying proof and inviting them to find the proof. The fact that he never revealed his proofs caused a lot of frustration among his contemporaries, and in his announcement of his famous last theorem he stated that he had a wonderful proof that was too large to include in the margin. He corresponded with Pascal and they did some early work on the mathematical rules of games of chance and early probability theory. He also did some early work on the calculus.

Fig. 7.3 Square numbers
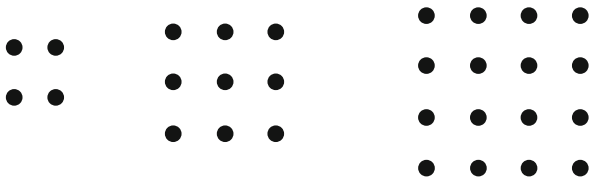
Fig. 7.4 Rectangular
numbers
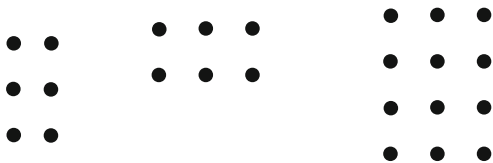
Fig. 7.5 Triangular numbers

$$n = 1 + 2 + \ldots + k$$
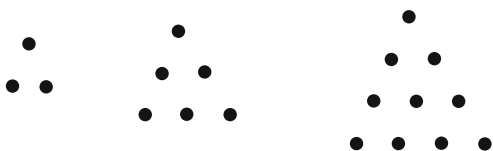
The Pythagoreans discovered the mathematical relationship between the harmony of music and numbers, and their philosophy was that numbers are hidden in everything from music to science and nature. This led to their philosophy that "*everything is number*".

7.2   Elementary Number Theory

A square number is an integer that is the square of another integer. For example, the number 4 is a square number since $4 = 2^2$. A number n is a square number if and only if one can arrange the n points in a square. For example, the square numbers 4, 9, 16 are represented in squares as shown in Fig. 7.3.

The square of an odd number is odd whereas the square of an even number is even. This is clear since an even number $n = 2k$ for some k and so $n^2 = 4k^2$ which is even.

A rectangular number n may be represented by a vertical and horizontal rectangle of n points. For example, the number 6 may be represented by a rectangle with length 3 and breadth 2, or a rectangle with length 2 and breadth 3 (Fig. 7.4).

A triangular number n may be represented by an equilateral triangle of n points (Fig. 7.5). A triangular number n is the sum of k natural numbers from 1 to

$k. =$ That is,

$$n = 1 + 2 + \cdots + k.$$

Parity of Integers The parity of an integer refers to whether the integer is odd or even. An integer n is odd if there is a remainder of 1 when it is divided by 2, and it is of the form $n = 2k + 1$. Otherwise, the number is even and of the form $n = 2k$.

The sum of two numbers is even if both are even or both are odd. The product of two numbers is even if at least one of the numbers is even. These properties are expressed as:

$$\text{even} \pm \text{even} = \text{even}$$

$$\text{even} \pm \text{odd} = \text{odd}$$

$$\text{odd} \pm \text{odd} = \text{even}$$

$$\text{even} \times \text{even} = \text{even}$$

$$\text{even} \times \text{odd} = \text{even}$$

$$\text{odd} \times \text{odd} = \text{odd}.$$

Divisors Let a and b be integers with $a \neq 0$ then a is said to be a divisor of b (denoted by $a|b$) if there exists an integer k such that $b = ka$.

A divisor of n is called a *trivial divisor* if it is either 1 or n itself, otherwise it is called a *non-trivial divisor*. A *proper divisor* of n is a divisor of n other than n itself.

Definition 7.1 (Prime Number) *A prime number is a number whose only divisors are trivial. There are an infinite number of prime numbers.*

The fundamental theorem of arithmetic states that every integer number can be factored as the product of prime numbers.

Mesenne Primes Mersenne primes are prime numbers of the form $2^p - 1$ where p is a prime. They are named after Marin Mersenne who was a seventeenth-century French monk, philosopher and mathematician (Fig. 7.6). There are 47 known Mersenne primes, and it remains an open question as to whether there are an infinite number of them.

Properties of Divisors

(i) $a|b$ and $a|c$ then $a|b + c$.
(ii) $a|b$ then $a|bc$.
(iii) $a|b$ and $b|c$ then $a|c$.

Proof (of Item i) Suppose $a|b$ and $a|c$ then $b = k_1 a$ and $c = k_2 a$.
   Then $b + c = (k_1 + k_2)a$ and so $a|b + c$.

Proof (of Item iii) Suppose $a|b$ and $b|c$ then $b = k_1 a$ and $c = k_2 b$.
   Then $c = k_2 b = (k_2 k_1)a$ and thus $a|c$.

Perfect and Amicable Numbers Perfect and amicable numbers have been studied for millennia. A positive integer m is said to be *perfect* if it is the sum of its proper

Fig. 7.6 Marin Mersenne

divisors. Two positive integers *m* and *n* are said to be an *amicable pair* if *m* is equal to the sum of the proper divisors of *n* and vice versa.

A *perfect number* is a number whose divisors add up to the number itself. For example, the number 6 is perfect since it has divisors 1, 2, 3 and $1 + 2 + 3 = 6$.

Perfect numbers are quite rare and Euclid showed that $2^p - 1(2^p - 1)$ is an even perfect number whenever $(2^p - 1)$ is prime. Euler later showed that all even perfect numbers are of this form. It is an open question as to whether there are any odd perfect numbers, and if such an odd perfect number *N* was to exist then $N > 10^{1500}$.

A prime number of the form $(2^p - 1)$ where *p* is prime is called a *Mersenne prime*. Each Mersenne prime generates an even perfect number and vice versa. That is, there is a one to one correspondence between the number of Mersenne primes and the number of even perfect numbers.

It remains an open question as to whether there are an infinite number of perfect numbers.

An *amicable pair* of numbers is a pair of numbers such that each number is the sum of divisors of the other number. For example, the numbers 220 and 284 are an amicable pair since the divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110, which have sum 284, and the divisors of 284 are 1, 2, 4, 71, 142, which have sum 220.

Theorem 7.1 (Division Algorithm) *For any integer a and any positive integer b there exists unique integers q and r such that*:

$$a = bq + r \quad 0 \le r < b.$$

Proof The first part of the proof is to show the existence of integers *q* and *r* such that the equality holds, and the second part of the proof is to prove uniqueness of *q* and *r*.

Consider $\ldots, -3b, -2b, -b, 0, b, 2b, 3b, \ldots$ then there must be an integer q such that

$$qb \leq a < (q+1)\,b.$$

Then $a - qb = r$ with $0 \leq r < b$ and so $a = bq + r$ and the existence of q and r is proved.

The second part of the proof is to show the uniqueness of q and r. Suppose q_1 and r_1 also satisfy $a = bq_1 + r_1$ with $0 \leq r_1 < b$ and suppose $r < r_1$. Then $bq + r = bq_1 + r_1$ and so $b(q - q_1) = r_1 - r$ and clearly $0 < (r_1 - r) < b$. Therefore, $b|(r_1 - r)$ which is impossible unless $r_1 - r = 0$. Hence, $r = r_1$ and $q = q_1$.

Theorem 7.2 (Irrationality of Square Root of 2) *The square root of 2 is an irrational number (i.e. it cannot be expressed as the quotient of two integer numbers).*

Proof The Pythagoreans[3] discovered this result and it led to a crisis in their community as number was considered to be the essence of everything in their world. The proof is indirect: i.e. the opposite of the desired result is assumed to be correct and it is showed that this assumption leads to a contradiction. Therefore, the assumption must be incorrect and so the result is proved.

Suppose $\sqrt{2}$ is rational then it can be put in the form p/q where p and q are integers and $q e 0$. Therefore, we can choose p, q to be co-prime (i.e. without any common factors) and so

$$(p/q)^2 = 2$$
$$\Rightarrow p^2/q^2 = 2$$
$$\Rightarrow p^2 = 2q^2$$
$$\Rightarrow 2|p^2$$
$$\Rightarrow 2|p$$
$$\Rightarrow p = 2k$$
$$\Rightarrow p^2 = 4k^2$$
$$\Rightarrow 4k^2 = 2q^2$$

---

[3] Pythagoras of Samos (a Greek island in the Aegean sea) was an influential ancient mathematician and philosopher of the sixth century B.C. He gained his mathematical knowledge from his travels throughout the ancient world (especially in Egypt and Babylon). He became convinced that everything is number and he and his followers discovered the relationship between mathematics and the physical world as well as relationships between numbers and music. On his return to Samos he founded a school and he later moved to Croton in southern Italy to set up a school. This school and the Pythagorean brotherhood became a secret society with religious beliefs such as reincarnation and they were focused on the study of mathematics. They maintained secrecy of the mathematical results that they discovered. Pythagoras is remembered today for Pythagoras's theorem, which states that for a right-angled triangle that the square of the hypotenuse is equal to the sum of the square of the other two sides. The Pythagorean's discovered the irrationality of the square root of two and as this result conflicted in a fundamental way with their philosophy that number is everything, and they suppressed the truth of this mathematical result.

$$\Rightarrow 2k^2 = q^2$$
$$\Rightarrow 2|q^2$$
$$\Rightarrow 2|q.$$

This is a contradiction as we have chosen p and q to be co-prime, and our assumption that there is a rational number that is the square root of 2 results in a contradiction. Therefore, this assumption must be false and we conclude that there is no rational number whose square is 2.

7.3 Prime Number Theory

A positive integer $n > 1$ is called prime if its only divisors are n and 1. A number that is not a prime is called composite.

Properties of Prime Numbers

1. There are an infinite number if primes.
2. There is a prime number p between n and $n! + 1$ such that $n < p \leq n! + 1$.
3. If n is composite then n has a prime divisor p such that $p \leq \sqrt{n}$.
4. There are arbitrary large gaps in the series of primes (given any $k > 0$ there exists k consecutive composite integers).

Proof (of Item i) Suppose there are a finite number of primes and they are listed as $p_1, p_2, p_3, \ldots, p_k$. Then consider the number N obtained by multiplying all known primes and adding 1. That is,

$$N = p_1 p_2 p_3 \ldots p_k + 1.$$

Clearly, N is not divisible by any of $p_1, p_2, p_3, \ldots, p_k$ since they all leave a remainder of 1. Therefore, N is either a new prime or divisible by a prime q (that is not in the list of $p_1, p_2, p_3, \ldots, p_k$).

This is a contradiction since the list was all primes, and so the assumption that there are a finite number of primes is false, and we deduce that there are an infinite number of primes.

Proof (of Item ii) Consider the integer $N = n! + 1$. If N is prime then we take $p = N$. Otherwise, N is composite and has a prime factor p. We will show that $p > n$.

Suppose, $p \leq n$ then $p|n!$ and since $p|N$ we have $p|n! + 1$ and therefore $p|1$, which is impossible. Therefore, $p > n$ and the result is proved.

Proof (of Item iii) Let p be the smallest prime divisor of n. Since n is composite $n = uv$ and clearly $p \leq u$ and $p \leq v$. Then $p^2 \leq uv = n$ and so $p \leq \sqrt{n}$.

Proof (of Item iv) Consider the k consecutive integers $(k+1)!+2, (k+1)!+3, \ldots, (k+1)!+k, (k+1)!+k+1$. Then each of these is composite since $j|(k+1)!+j$ where $2 \leq j \leq k+1$.

Fig. 7.7 Primes between
1 and 50

	2	3	4	5	6	7	8		10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Algorithm for Determining Primes The *sieve of Eratosthenes algorithm* is a famous algorithm for determining the prime numbers up to a given number n. The Hellenistic mathematician, Eratosthenes, developed it.[4]

The algorithm involves first listing all of the numbers from 2 to n. The first step is to remove all multiples of 2 up to n, the second step is to remove all multiples of 3 up to n and so on.

The kth step involves removing multiples of the kth prime p_k up to n and the steps in the algorithm continue while $p \leq \sqrt{n}$. The numbers remaining in the list are the prime numbers from 2 to n.

1. List the integers from 2 to n.
2. For each prime p_k up to $\sqrt{n}$ remove all multiples of p_k.
3. The numbers remaining are the prime numbers between 2 and n.

The list of primes between 1 and 50 are given in Fig. 7.7.

Theorem 7.3 (Fundamental Theorem of Arithmetic) *Every natural number $n > 1$ may be written uniquely as the product of primes*:

$$n = p_1{}^{\alpha_1} p_2{}^{\alpha_2} p_3{}^{\alpha_3} \ldots p_k{}^{\alpha_k}.$$

Proof There are two parts to the proof. The first part shows that there is a factorisation and the second part shows that the factorisation is unique.

Part(a) If n is prime then it is a product with a single prime factor. Otherwise, n can be factored into the product of two numbers ab where $a > 1$ and $b > 1$. The argument can then be applied to each of a and b each of which is either prime or can be factored as the product of two numbers both of which are greater than 1. Continue in this way with the numbers involved decreasing with every step in the process until eventually all of the numbers must be prime.

Part(b) Suppose the factorisation is not unique and let $n > 1$ be the smallest number that has more than one factorisation of primes. Then n may be expressed as follows:

$$n = p_1 p_2 p_3 \ldots p_k = q_1 q_2 q_3 \ldots q_r.$$

Clearly, $k > 1$ and $r > 1$ and $p_i \neq q_j$ for $(i = 1, \ldots, k)$ and $(j = 1, \ldots, r)$ as otherwise we could construct a number smaller than n (e.g. n/p_i where $p_i = q_j$)

---

[4] Eratosthenes also determined an approximation of the circumference of the earth.

that has two distinct factorisations. Next, without loss of generality take $p_1 < q_1$ and define the number N by:

$$N = (q_1 - p_1) q_2 q_3 \ldots q_r$$
$$= p_1 p_2 p_3 \ldots p_k - p_1 q_2 q_3 \ldots q_r$$
$$= p_1 (p_2 p_3 \ldots p_k - q_2 q_3 \ldots q_r).$$

Clearly $1 < N < n$ and so N is uniquely factorisable into primes. However, clearly p_1 is not a divisor of $(q_1 - p_1)$, and so N has two distinct factorisations, which is a contradiction of the choice of n.

7.3.1   Greatest Common Divisors (GCD)

Let a and b be integers not both zero. The *greatest common divisor* d of a and b is a divisor of a and b (i.e. $d|a$ and $d|b$), and it is the largest such divisor (i.e. if $k|a$ and $k|b$ then $k|d$). It is denoted by $\gcd(a, b)$.

Properties of Greatest Common Divisors

(i) Let a and b be integers not both zero then exists integers x and y such that:
     $d = \gcd(a, b) = ax + by$.
(ii) Let a and b be integers not both zero then the set $S = \{ax + by$ where $x, y \in \mathbb{Z}\}$
     is the set of all multiples of $d = \gcd(a, b)$.

Proof (of Item i) Consider the set of all linear combinations of a and b forming the set $\{ka + nb : k, n \in \mathbb{Z}\}$. Clearly, this set includes positive and negative numbers. Choose x and y such that $m = ax + by$ is the smallest positive integer in the set. Then we shall show that m is the greatest common divisor.

We know from the division algorithm that $a = mq + r$ where $0 \le r < m$. Thus

$$r = a - mq = a - (ax + by)q = (1 - qx)a + (- yq)b,$$

r is a linear combination of a and b and so r must be 0 from the definition of m. Therefore, $m|a$ and similarly $m|b$ and so m is a common divisor of a and b. Since, the greatest common divisor d is such that $d|a$ and $d|b$ and $d \le m$ we must have $d = m$.

Proof (of Item ii) This follows since $d|a$ and $d|b \implies d|ax + by$ for all integers x and y and so every element in the set $S = \{ax + by$ where $x, y \in \mathbb{Z}\}$ is a multiple of d.

Relatively Prime Two integers a, b are relatively prime if $\gcd(a, b) = 1$.

Properties If p is a prime and $p|ab$ then $p|a$ or $p|b$.

Proof Suppose $p \nmid a$ then from the results on the greatest common divisor we have $\gcd(a, p) = 1$. That is,

$$ra + sp = 1$$
$$\Rightarrow rab + spb = b$$
$$\Rightarrow p|b \text{ (since } p|rab \text{ and } p|spb \text{ and so } p|rab + spb).$$

7.3.2   Least Common Multiple (LCM)

If m is a multiple of a and m is a multiple of b then it is said to be a *common multiple* of a and b. The least common multiple is the smallest of the common multiples of a and b and it is denoted by lcm (a, b).

Properties If x is a common multiple of a and b then $m|x$. That is, every common multiple of a and b is a multiple of the least common multiple m.

Proof We assume that both a and b are non-zero as otherwise the result is trivial since all common multiples are 0. Clearly, by the division algorithm we have:

$$x = mq + r, \quad \text{where } 0 \leq r < m.$$

Since x is a common multiple of a and b we have $a|x$ and $b|x$ and also that $a|m$ and $b|m$. Therefore, $a|r$ and $b|r$. and so r is a common multiple of a and b and since m is the least common multiple we have r is 0. Therefore x is a multiple of the least common multiple m as required.

7.3.3   Euclid's Algorithm

Euclid's[5] algorithm is one of the oldest known algorithms and it provides a procedure for finding the greatest common divisor of two numbers. It appears in Book VII of Euclid's *Elements*, and the algorithm was known prior to Euclid (Fig. 7.8).

Lemma *Let a, b, q and r be integers with $b > 0$ and $0 \leq r < b$ such that $a = bq + r$. Then $\gcd(a, b) = \gcd(b, r)$.*

Proof Let $K = \gcd(a, b)$ and let $L = \gcd(b, r)$ and we therefore need to show that $K = L$. Suppose m is a divisor of a and b then as $a = bq + r$ we have m is a divisor of r and so any common divisor of a and b is a divisor of r.

Similarly, any common divisor n of b and r is a divisor of a. Therefore, the greatest common divisor of a and b is equal to the greatest common divisor of b and r.

---

[5] Euclid was a third century B.C. Hellenistic mathematician and is considered the father of geometry.

Fig. 7.8 Euclid of Alexandria

Theorem 7.4 (Euclid's Algorithm)   *Euclid's algorithm for finding the greatest common divisor of two positive integers a and b involves applying the division algorithm repeatedly as follows*:

$$
\begin{aligned}
a &= bq_0 + r_1 & 0 < r_1 < b \\
b &= r_1 q_1 + r_2 & 0 < r_2 < r_1 \\
r_1 &= r_2 q_2 + r_3 & 0 < r_3 < r_2 \\
& \quad \cdots \\
& \quad \cdots \\
r_{n-2} &= r_{n-1} q_{n-1} + r_n & 0 < r_n < r_{n-1} \\
r_{n-1} &= r_n q_n.
\end{aligned}
$$

*Then r_n (i.e. the last non-zero remainder) is the greatest common divisor of a and b:
i.e.* $\gcd(a, b) = r_n$.

Proof   It is clear from the construction that r_n is a divisor of $r_{n-1}, r_{n-2}, \ldots, r_3, r_2, r_1$ and of a and b. Clearly, any common divisor of a and b will also divide r_n. Using the results from the lemma above we have:

$$
\begin{aligned}
\gcd(a, b) & \\
&= \gcd(b, r_1) \\
&= \gcd(r_1 r_2) \\
&= \cdots \\
&= \gcd(r_{n-2} r_{n-1}) \\
&= \gcd(r_{n-1}, r_n) \\
&= r_n.
\end{aligned}
$$

Lemma *Let n be a positive integer greater than* 1 *then the positive divisors of n are precisely those integers of the form*:

$$d = p_1^{\beta_1} p_2^{\beta_2} p_3^{\beta_3} \ldots p_k^{\beta_k} \quad (\text{where } 0 \leq \beta_i \leq \alpha_i),$$

where the unique factorisation of n is given by:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} p_3^{\alpha_3} \ldots p_k^{\alpha_k}.$$

Proof Suppose d is a divisor of n then $n = dq$. By the unique factorisation theorem the prime factorisation of n is unique, and so the prime numbers in the factorisation of d must appear in the prime factors p_1, p_2, $p_3, \ldots, p_k$ of n.

Clearly, the power β_i of p_i must be less than or equal to α_i: i.e. $\beta_i \leq \alpha_i$. Conversely, whenever $\beta_i \leq \alpha_i$ then clearly d divides n.

7.3.4   *Distribution of Primes*

We have already shown that there are an infinite number of primes. However, most integer numbers are composite and a reasonable question to ask is how many primes are there less than a certain number. The number of primes less than or equal to x is known as the prime distribution function (denoted by $\pi(x)$) and it is defined by:

$$\pi(x) = \sum_{p \leq x} 1 \quad (\text{where } p \text{ is prime}).$$

The prime distribution function satisfies the following properties:

(i) $\lim\limits_{x \to \infty} \frac{\pi(x)}{x} = 0.$

(ii) $\lim\limits_{x \to \infty} \pi(x) = \infty.$

The first property expresses the fact that most integer numbers are composite, and the second property expresses the fact that there are an infinite number of prime numbers.

There is an approximation of the prime distribution function in terms of the logarithmic function $(x / \ln x)$ as follows:

$$\lim_{x \to \infty} \frac{\pi(x)}{x / \ln x} = 1 \quad (\text{prime number theorem})$$

The approximation $x / \ln x$ to $p(x)$ gives an easy way to determine the approximate value of $p(x)$ for a given value of x. This result is known as the *prime number theorem*, and Gauss originally conjectured this theorem.

Palindromic Primes A *palindromic prime* is a prime number that is also a palindrome (i.e. it reads the same left to right as right to left). For example, 11, 101, 353 are all palindromic primes.

All palindromic primes (apart from 11) have an odd number of digits. It is an open question as to whether there are an infinite number of palindromic primes.

Let $\sigma(m)$ denote the sum of all the positive divisors of m (including m):

$$\sigma(m) = \sum_{d \mid m} d.$$

Let $s(m)$ denote the sum of all the positive divisors of m (excluding m):

$$s(m) = \sigma(m) - m.$$

Clearly, $s(m) = m$ and $\sigma(m) = 2m$ when m is a perfect number.

Theorem 7.5 (Euler Euclid Theorem)   *The positive integer n is an even perfect number if and only if $n = 2^{p-1}(2^p - 1)$ where $2^p - 1$ is a Mersenne prime.*

Proof Suppose $n = 2^{p-1}(2^p - 1)$ where $2^p - 1$ is a Mersenne prime then:

$$\sigma(n) = \sigma(2^{p-1}(2^p - 1))$$

$$= \sigma(2^{p-1})\sigma(2^p - 1)$$

$$= \sigma(2^{p-1})2^p \quad (2^p - 1 \text{is prime with 21 and itself})$$

$$= (2^p - 1)2^p \quad (\text{sum of airthmetic series})$$

$$= (2^p - 1)2.2^{p-1}$$

$$= 2.2^{p-1}(2^p - 1)$$

$$= 2n.$$

Therefore, n is a perfect number since $\sigma(n) = 2n$.

The next part of the proof is to show that any even perfect number must be of the form above. Let n be an arbitrary even perfect number then $n = 2^{p-1}q$ with q odd and so the gcd $(2^{p-1}, q) = 1$ and so:

$$\sigma(n) = \sigma(2^{p-1}q)$$

$$= \sigma(2^{p-1})\sigma(q)$$

$$= (2^p - 1)\sigma(q)$$

$$\sigma(n) = 2n \quad (\text{since } n \text{ is perfect})$$

$$= 2.2^{p-1}q$$

$$= 2^p q. \tag{7.1}$$

Fig. 7.9 Leonard Euler

Therefore,

$$2^p q$$
$$= (2^p - 1)\sigma(q)$$
$$= (2^p - 1)(s(q) + q)$$
$$= (2^p - 1)s(q) + (2^p - 1)q$$
$$= (2^p - 1)s(q) + 2^p q - q.$$

Therefore, $(2^p - 1)s(q) = q$.

Therefore, $d = s(q)$ is a proper divisor of q. However, $s(q)$ is the sum of all the proper divisors of q including d, and so d is the only proper divisor of q and $d = 1$. Therefore, $q = (2^p - 1)$ is a Mersenne prime.

Euler ϕ Function The Euler[6] ϕ function (also known as the *totient function*) is defined for a given positive integer n to be the number of positive integers k less than n that are relatively prime to n (Fig. 7.9). Two integers a, b are relatively prime if $\gcd(a, b) = 1$,

$$\varphi(n) = \sum_{1 \leq k < n} 1, \quad \text{where } \gcd(k, n) = 1.$$

---

[6] Euler was an eighteenth-century Swiss mathematician who made important contributions to mathematics and physics. His contributions include graph theory (e.g. the well-known formula $V - E + F = 2$), calculus, infinite series, the exponential function for complex numbers and the totient function.

7.4   Theory of Congruences[7]

Let a be an integer and n a positive integer greater than 1 then $(a \bmod n)$ is defined to be the remainder r when a is divided by n. That is,

$$a = kn + r, \quad \text{where } 0 \leq r < n.$$

Definition Suppose a, b are integers and n a positive integer then a is said to be congruent to b modulo n denoted by $a \equiv b \pmod{n}$ if they both have the same remainder when divided by n.

This is equivalent to n being a divisor of $(a - b)$ or $n|(a - b)$ since we have $a = k_1 n + r$ and $b = k_2 n + r$ and so $(a - b) = (k_1 - k_2)n$ and so $n|(a - b)$.

Theorem 7.6 *Congruence modulo n is an equivalence relation on the set of integers*: *i.e. it is a reflexive, symmetric and transitive relation.*

Proof

1. Reflexive
   For any integer a it is clear that $a \equiv a \pmod{n}$ since $a - a = 0.n$
2. Symmetric
   Suppose $a \equiv b \pmod{n}$ then $a - b = kn$. Clearly, $b - a = -kn$ and so $b \equiv a \pmod{n}$.
3. Transitive
   Suppose $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$

$$\Rightarrow a - b = k_1 n \text{ and } b - c = k_2 n$$

$$\Rightarrow a - c = (a - b) + (b - c)$$

$$= k_1 n + k_2 n$$

$$= (k_1 + k_2)n$$

$$\Rightarrow a \equiv c \pmod{n}.$$

Therefore, congruence modulo n is an equivalence relation, and an equivalence relation partitions a set S into equivalence classes. The integers are partitioned into n equivalence classes for the congruence modulo n equivalence relation, and these are called *congruence classes* or *residue classes*.

The residue class of a modulo n is denoted by $[a]_n$ or just $[a]$ when n is clear. It is the set of all those integers that are congruent to a modulo n,

$$[a]_n = \{x : x \in \mathbb{Z} \text{ and } x \equiv a \pmod{n}\} = \{a + kn : k \in \mathbb{Z}\}.$$

---

[7] The theory of congruences was introduced by the German mathematician, Carl Friedrich Gauss.

Any two equivalence classes $[a]$ and $[b]$ are either equal or disjoint: i.e. we have $[a] = [b]$ or $[a] \cap [b] = \varnothing$. The set of all residue classes modulo n is denoted by:

$$\mathbb{Z}/n\mathbb{Z} = \mathbb{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\} = \{[0]_n, [1]_n, \ldots, [n - 1]_n\}.$$

For example, consider $\mathbb{Z}_4$ the residue classes mod 4 then

$$[0]_4 = \{\ldots, -8, -4, 0, 4, 8, \ldots\}$$
$$[1]_4 = \{\ldots, -7, -3, 1, 5, 9, \ldots\}$$
$$[2]_4 = \{\ldots, -6, -2, 2, 6, 10, \ldots\}$$
$$[3]_4 = \{\ldots, -5, -1, 3, 7, 11, \ldots\}.$$

The *reduced residue class* is a set of integers r_i such that $(r_i, n) = 1$ and r_i is not congruent to $r_j \pmod{n}$ for $i \neq j$, and such that every x relatively prime to n is congruent modulo n to for some element r_i of the set. There are $\phi(n)$ elements $\{r_1, r_2, \ldots, r_{\phi(n)}\}$ in the reduced residue class set S.

Modular Arithmetic Addition, subtraction and multiplication may be defined in $\mathbb{Z}/n\mathbb{Z}$ and are similar to these operations in $\mathbb{Z}$. Given a positive integer n and integers a, b, c, d such that $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$ then the following are properties of modular arithmetic:

(i) $a + c \equiv b + d \pmod{n}$ and $a - c \equiv b - d \pmod{n}$.
(ii) $ac \equiv bd \pmod{n}$.
(iii) $a^m \equiv b^m \pmod{n} \quad \forall m \in \mathbb{N}$.

Proof (of Item ii) Let $a = kn + b$ and $c = ln + d$ for some $k, l \in \mathbb{Z}$ then

$$ac = (kn + b)(ln + d)$$
$$= (kn)(ln) + (kn)d + b(ln) + bd$$
$$= (knl + kd + bl)n + bd$$
$$= sn + bd \quad (\text{where } s = knl + kd + bl)$$

and so

$$ac \equiv bd \pmod{n}.$$

The three properties above may be expressed in the following equivalent formulation:

(i) $[a + c]_n = [b + d]_n$ and $[a - c]_n = [b - d]_n$.
(ii) $[ac]_n = [bd]_n$.
(iii) $[a^m]_n = [b^m]_n \quad \forall m \in N$.

Two integers x, y are said to be multiplicative inverses of each other modulo n if:

$$xy \equiv 1 \pmod{n}.$$

However, x does not always have an inverse modulo n, as $[3]_6 . [2]_6 = [0]_6$, and so $[3]_6$ does not have a multiplicative inverse (mod 6). However, if n and x are relatively

prime then it is easy to see that x has an inverse (mod n) since we know that there are integers k, l such that $kx + ln = 1$.

Given $n > 0$ there are $\varphi(n)$ numbers b that are relatively prime to n and so there are $\varphi(n)$ numbers that have an inverse modulo n. Therefore, for p prime there are $p-1$ elements that have an inverse (mod p).

Theorem 7.7 (Euler's Theorem) *Let a and n be positive integers with* $\gcd(a,n) = 1$. *Then*

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

Proof Let $\{r_1, r_2, \ldots, r_{\varphi(n)}\}$ be the reduced residue system (mod n). Then $\{ar_1, ar_2, \ldots, ar_{\varphi(n)}\}$ is also a reduced residue system (mod n) since $ar_i \equiv ar_j \pmod{n}$ and $(a,n) = 1$ implies that $r_i \equiv r_j \pmod{n}$.

For each r_i there is exactly one r_j such that $ar_i \equiv ar_j \pmod{n}$, and different r_i will have different corresponding ar_j. Therefore, $\{ar_1, ar_2, \ldots, ar_{j(n)}\}$ are just the residues module n of $\{r_1, r_2, \ldots, r_{j(n)}\}$ but not necessarily in the same order. Multiplying we get:

$$\prod_{j=1}^{\varphi(n)} (ar_j) \equiv \prod_{i=1}^{\varphi(n)} r_i \pmod{n},$$

$$a^{\phi(n)} \prod_{j=1}^{\varphi(n)} (r_j) \equiv \prod_{i=1}^{\varphi(n)} r_i \pmod{n}.$$

Since $(r_j, n) = 1$ we can deduce that $a^{\phi(n)} \equiv 1 \pmod{n}$ from the result that $ax \equiv ay \pmod{n}$ and $(a,n) = 1$ then $x \equiv y \pmod{n}$.

Theorem 7.8 (Fermat's Little Theorem) *Let a be a positive integer and p a prime. If* $\gcd(a,p) = 1$ *then*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof This result is an immediate corollary to Euler's theorem as $\varphi(p) = p - 1$.

Theorem 7.9 (Wilson's Theorem) *If p is a prime then* $(p-1)! \equiv -1 \pmod{p}$.

Proof Each element $a \in 1, 2, \ldots, p$ - 1 has an inverse a^{-1} such that $aa^{-1} \equiv 1 \pmod{p}$. Exactly two of these elements 1 and $p-1$ are their own inverse (i.e. $x^2 \equiv 1 \pmod{p}$ has two solutions 1 and $p-1$). Therefore, the product $1 \cdot 2 \cdots p - 1 \pmod{p} = p - 1 \pmod{p} \equiv -1 \pmod{p}$.

Diophantine Equations The word "*Diophantine*" is derived from the name of the third-century mathematician, Diophantus, who lived in the city of Alexandria in Egypt. Diophantus studied various polynomial equations of the form $f(x, y, z, \ldots) = 0$ with integer coefficients to determine which of them had integer solutions.

A Diophantine equation may have no solution, a finite number of solutions or an infinite number of solutions. The integral solutions of a Diophantine equation $f(x, y) = 0$ may be interpreted geometrically as the points on the curve with integral coordinates.

Example A linear Diophantine equation $ax + by = c$ is an algebraic equation with two variables x and y, with integer solutions for x and y.

7.5 Review Questions

1. Show that
   a. If $a|b$ then $a|bc$.
   b. If $a|b$ then $c|d$ then $ac|bd$.
2. Show that 1184 and 1210 are an amicable pair.
3. Use the Euclidean algorithm to find $g = \gcd(b, c)$ where $b = 42823$ and $c = 6409$, and find integers x and y such that $bx + cy = g$.
4. List all integers x in the range $1 \le x \le 100$ such that $x \equiv 7 \pmod{17}$.
5. Evaluate $\phi(m)$ for $m = 1, 2, 3, \ldots, 12$.
6. Determine a complete residue system modulo 12 and a reduced residue system modulo 12.

7.6 Summary

Number theory is the branch of mathematics that is concerned with the mathematical properties of the natural numbers and integers. These include properties such as, whether a number is prime or composite, the prime factors of a number, the greatest common divisor and least common multiple of two numbers and so on.

The natural numbers $\mathbb{N}$ consist of the numbers $\{1, 2, 3, \ldots\}$. The integer numbers $\mathbb{Z}$ consist of $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The rational numbers $\mathbb{Q}$ consist of all numbers of the form $\{p/q$ where p and q are integers and $q \neq 0\}$. Number theory has been applied to cryptography in the computing field.

Prime numbers have no factors apart from themselves and 1, and there are an infinite number of primes. The sieve of Eratosthenes algorithm may be employed to determine prime numbers, and the approximation to the number of prime numbers less than a specific number n is given by the prime distribution function $\pi(n) = n/\ln n$. Prime numbers are the key building blocks in number theory, and the fundamental theorem of arithmetic states that every number may be written uniquely as the product of factors of prime numbers.

Mersenne primes and perfect numbers were considered and it was shown that there is a one to one correspondence between the Mersenne primes and the even perfect numbers.

Modulo arithmetic including addition, subtraction and multiplication were defined, and the residue classes and reduced residue classes were discussed.

There are several unsolved problems in number theory. These include Goldbach's conjecture that states that every even integer is the sum of two primes. Other open questions include whether there are an infinite number of Mersenne primes and palindromic primes.

Chapter 8
Cryptography

Key Topics
    Caesar Cipher
    Enigma Codes
    Bletchley Park
    Turing
    Public and Private Keys
    Symmetric Keys
    Block Ciphers
    RSA

8.1 Introduction

Cryptography was originally employed to protect communication of private information between individuals. Today, it consists of mathematical techniques that provide secrecy in the transmission of messages between computers, and its objective is to solve security problems such as privacy and authentication over a communications channel.

It involves enciphering and deciphering messages, and theoretical results from number theory are employed to convert the original message (or plaintext) into ciphertext that is then transmitted over a secure channel to the intended recipient. The ciphertext is meaningless to anyone other than the intended recipient, and the recipient uses a key to decrypt the received ciphertext and to obtain the original message.

The origin of the word "cryptography" is from the Greek '*kryptos*' meaning hidden, and '*graphein*' meaning to write. The field of cryptography is concerned with techniques by which information may be concealed in ciphertexts and made unintelligible to all but the intended recipient. This ensures the privacy of the information sent, as any information intercepted will be meaningless to anyone other than the recipient.

Fig. 8.1 Caesar cipher

Alphabet Symbol	abcde fghij klmno pqrst uvwxyz
Cipher Symbol	dfegh ijklm nopqr stuvw xyzabc

Julius Caesar developed one of the earliest ciphers on his military campaigns in Gaul. His objective was to communicate important messages safely to his generals. This is one of the simplest and widely known encryption techniques, and it involves the substitution of each letter in the plaintext (i.e. the original message) by a letter a fixed number of positions down in the alphabet. The Caesar cipher involves a shift of three positions and this leads to the letter B being replaced by E, the letter C by F, and so on.

The Caesar cipher is easily broken, as the frequency distribution of letters may be employed to determine the mapping. However, the Gaulish tribes were mainly illiterate, and the cipher is likely to have provided good security. The translation of the Roman letters by the Caesar cipher (with a shift key of 3) can be seen in Fig. 8.1.

The process of enciphering a message (i.e. the plaintext) simply involves going through each letter in the plaintext and writing down the corresponding cipher letter. The enciphering of the plaintext message "summer solstice" involves the following:

Plaintext: Summer Solstice

Cipher Text: vxpphu vrovwleh

The process of deciphering a cipher message involves doing the reverse operation: i.e. for each cipher letter the corresponding plaintext letter is identified from the table:

Cipher Text: vxpphu vrovwleh

Plaintext: Summer Solstice

The encryption may also be represented using modular arithmetic. This involves using the numbers 0–25 to represent the alphabet letters, and the encryption of a letter is given by a shift transformation of three (modulo 26). This is simply addition (modula 26): i.e. the encoding of the plaintext letter x is given by:

$$c = x + 3 \,(\mathrm{mod}\ 26).$$

Similarly, the decoding of the cipher letter c is given by:

$$x = c - 3 \,(\mathrm{mod}\ 26).$$

The Caesar cipher was still in use up to the early twentieth century. However, by then frequency analysis techniques were available to break the cipher. The Vignère cipher uses a Caesar cipher with a different shift at each position in the text. The value of the shift to be employed with each plaintext letter is defined using a repeating keyword.

Fig. 8.2 The Enigma
machine

8.2   Breaking the Enigma Codes

The Enigma codes were used by the Germans during the second world war for the secure transmission of naval messages to their submarines. These messages contained top-secret information on German submarine and naval activities in the Atlantic, and the threat that they posed to British and Allied shipping.

The codes allowed messages to be passed secretly using encryption, and any unauthorised interception was meaningless to the Allies. The plaintext (i.e. the original message) was converted by the Enigma machine (Fig. 8.2) into the encrypted text, and these messages were then transmitted by the Germans to their submarines in the Atlantic or to their bases throughout Europe.

The Enigma cipher was invented in 1918 and the Germans believed it to be unbreakable. A letter was typed in German into the machine, and electrical impulses through a series of rotating wheels and wires produced the encrypted letter which was lit up on a panel above the keyboard. The recipient typed the received message into his machine and the decrypted message was lit up letter by letter above the keyboard. The rotors and wires of the machine could be configured in many different ways, and during the war the cipher settings were changed at least once a day. The odds against anyone breaking the Enigma machine without knowing the setting were extremely low (150×10^{18} to 1).

The British code and cipher school was relocated from London to Bletchley Park at the start of the second world war. It was located in the town of Bletchley near Milton Keynes (about 50 miles north west of London). It was commanded by Alistair Dennison and was known as Station X, and during the second world war there were thousands of people working there. The team at Bletchley Park broke the Enigma codes, and therefore made vital contributions to the British and Allied war effort (Fig. 8.3).

Polish cryptanalysts did important work in breaking the Enigma machine in the early 1930s, and they constructed a replica of the machine. They passed their knowledge on to the British and gave them the replica just prior to the German invasion of Poland. The team at Bletchley built the Polish work, and the team included Alan Turing[1] (Fig. 8.4) and other mathematicians.

---

[1] Turing made fundamental contributions to computing, including the theoretical Turing machine.

Fig. 8.3 Bletchley Park

Fig. 8.4 Alan Turing

The code-breaking teams worked in various huts in Bletchley Park. Hut 6 focussed on air force and army cyphers, and hut 8 focussed on naval cyphers. The deciphered messages were then converted into intelligence reports, with air force and army intelligence reports produced by the team in hut 3, and naval intelligence reports produced by the team in hut 4. The raw material (i.e. the encrypted messages) to be deciphered came from wireless intercept stations dotted around Britain, and from various countries overseas. These stations listened to German radio messages, and sent them to Bletchley Park to be deciphered and analysed.

Turing devised a machine to assist with breaking the codes (an idea that was originally proposed by the Polish cryptanalysts). This electromechanical machine was known as the bombe, and its goal was to find the right settings of the Enigma machine for that particular day. The machine greatly reduced the odds and the time required to determine the settings on the Enigma machine, and it became the main tool for reading the Enigma traffic during the war. The bombe was first installed in early 1940 and it weighed over a ton (Fig. 8.5). It was named after a cryptological device designed in 1938 by the Polish cryptologist, Marian Rejewski.

A standard Enigma machine employed a set of rotors, and each rotor could be in any of 26 positions. The bombe tried each possible rotor position and applied a test. The test eliminated almost all of the positions and left a smaller number of cases to be dealt with. The test required the cryptologist to have a suitable "*crib*": i.e. a section of ciphertext for which he could guess the corresponding plaintext.

Fig. 8.5 Replica of bombe
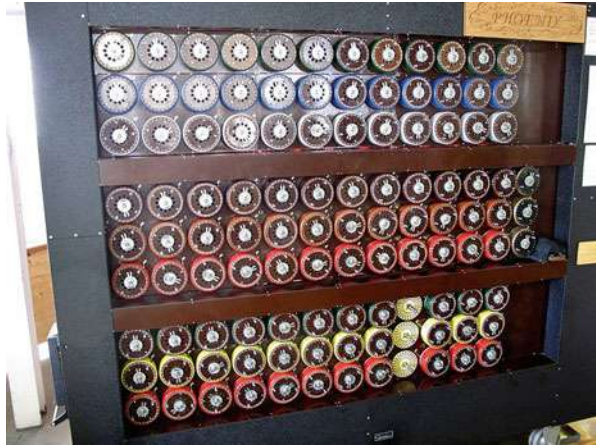
For each possible setting of the rotors, the bombe employed the crib to perform a chain of logical deductions. The bombe detected when a contradiction had occurred and it then ruled out that setting and moved onto the next. Most of the possible settings would lead to contradictions and could then be discarded. This would leave only a few settings to be investigated in detail.

One of the earliest computers, the Colossus, was developed by Tommy Flowers and others at Bletchley Park to decipher the important Lorenz codes transmitted by German high command to their generals in the field.

The site at Bletchley Park was used for training purposes after the second world war. The Government Communication Headquarters (GCHQ) was its successor, and it is now based in Cheltenham.

The codebreakers who worked at Bletchley Park were required to remain silent about their achievements until the mid-1970s when the wartime information was declassified. The link between British Intelligence and Bletchley Park came to an end in the mid-1980s.

It was decided in the mid-1990s to restore Bletchley Park, and today it is run as a museum by the Bletchley Park Trust. There is more information about Bletchley Park in [ORg:12].

8.3 Cryptographic Systems

A cryptographic system is a computer system that is concerned with the secure transmission of messages. The message is encrypted prior to its transmission, which ensures that any unauthorised interception and viewing of the message is meaningless to anyone other than the intended recipient. The recipient uses a key to decrypt the ciphertext, and to retrieve the original message.

There are essentially two different types of cryptographic systems employed, and these are public key cryptosystems and secret key cryptosystems. A *public key*

Table 8.1 Notation in
cryptography

Symbol	Description
M	Represents the message (plaintext)
C	Represents the encrypted message (ciphertext)
e_k	Represents the encryption key
d_k	Represents the decryption key
E	Represents the encryption process
D	Represents the decryption process

cryptosystem is an asymmetric cryptosystem where two different keys are employed:
one for encryption and one for decryption. The fact that a person is able to encrypt
a message does not mean that the person is able to decrypt a message.

In a *secret key cryptosystem* the same key is used for both encryption and de-
cryption. Anyone who has knowledge on how to encrypt messages has sufficient
knowledge to decrypt messages. The notation in Table 8.1 is employed.

The encryption and decryption algorithms satisfy the following equation:

$$Dd_k(C) \; = Dd_k(Ee_k(M)) \; = M.$$

There are two different keys employed in a public key cryptosystem. These are the
encryption key e_k and the decryption key d_k with $e_k \neq d_k$. It is called asymmetric
since the encryption key differs from the decryption key.

There is just one key employed in a secret key cryptosystem, with the same key e_k
is used for both encryption and decryption. It is called *symmetric* since the encryption
key is the same as the decryption key: i.e. $e_k = d_k$.

8.4   Symmetric Key Systems

The same secret key is employed for encryption and decryption in a symmetric key
cryptosystem (Fig. 8.6). The sender and the receiver first agree a shared key prior
to communication, and this is done over a secure channel to ensure that the shared
key remains secret. They can then begin to encrypt and decrypt messages using the
secret key. Anyone who is able to encrypt a message has sufficient information to
decrypt the message.

The encryption of a message is in effect a transformation from the space of mes-
sages M to the space of cryptosystems $\mathbb{C}$. That is, the encryption of a message with
key k is an invertible transformation f such that:

$$f : \mathsf{M} \xrightarrow{k} \mathbb{C}$$

The ciphertext is given by $C = E_k(M)$ where $M \in \mathsf{M}$ and $C \in \mathbb{C}$. The legitimate
receiver of the message knows the secret key k (as it will have transmitted previously
over a secure channel), and so the ciphertext C can be decrypted by the inverse
transformation f^{-1} defined by:

$$f^{-1} : \mathbb{C} \xrightarrow{k} \mathsf{M}$$

Fig. 8.6 Symmetric key cryptosystem
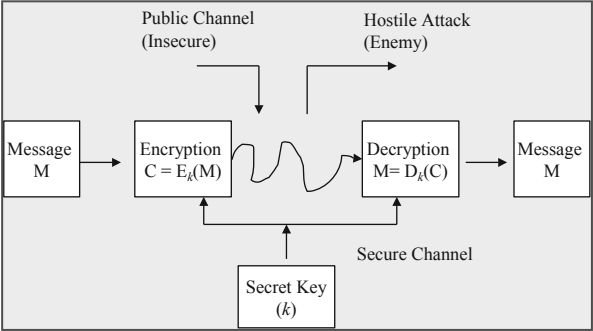
Table 8.2 Advantages and disadvantages of symmetric key systems

Advantages	Disadvantages
Encryption process is simple (as the same key is used for encryption and decryption)	A shared key must be agreed between two parties
It is faster than public key systems	Key exchange is difficult as there needs to be a secure channel between the two parties (to ensure that the key remains secret)
It uses less computer resources than public key systems	If a user has n trading partners then n secret keys must be maintained (one for each partner)
It uses a different key for communication with every different party	There are problems with the management and security of all of these keys (due to volume of keys that needs to be maintained)
	Authenticity of origin or receipt cannot be proved (as key is shared)

Therefore, we have that $D_k(C) = D_k(E_k(M))$ the original plaintext message.

There are advantages and disadvantages of symmetric key systems, and these are included in Table 8.2.

Examples of Symmetric Key Systems

(i) *Caesar Cipher*

The Caesar cipher may be defined using modular arithmetic. It involves a shift of three places for each letter in the plaintext, and the alphabetic letters are represented by the numbers 0–25. The encryption is carried out by addition (modula 26). The encryption of the plaintext letter x is given by:

$$c = x + 3 \pmod{26}$$

Similarly, the decryption of a cipher letter c is given by:

$$x = c - 3 \pmod{26}$$

(ii) *Generalised Caesar Cipher*

This is a generalisation of the Caesar cipher to a shift of k (the Caesar cipher involves a shift of three). This is given by:

$$f_k = E_k(x) \equiv x + k \,(\text{mod } 26) \quad 0 \le k \le 25$$

$$f_k^{-1} = D_k(c) \equiv c - k \,(\text{mod } 26) \quad 0 \le k \le 25$$

(iii) *Affine Transformation*

This is a more general transformation and is defined by:

$$f_{(a,b)} = E_{(a,b)}(x) \equiv ax + b \,(\text{mod } 26) \quad 0 \le a, b, x \le 25 \text{ and } \gcd(a, 26) = 1$$

$$f^{-1}{}_{(a,b)} = D_{(a,b)}(c) \equiv a^{-1}(c - b)(\text{mod } 26) \quad a^{-1} \text{ is the inverse of } a \text{ mod } 26$$

(iv) *Block Ciphers*

Stream ciphers encrypt a single letter at a time and are easy to break. Block ciphers offer greater security, and the plaintext is split into groups of letters, and the encryption is performed on the block of letters rather than on a single letter.

The message is split into blocks of n letters: $M_1, M_2, \ldots M_k$ where each $M_i (1 \le i \le k)$ is a block n letters. The letters in the message are translated into their numerical equivalents, and the ciphertext is formed as follows:

$$C_i \equiv AM_i + B \,(\text{mod } N) \quad i = 1, 2, \ldots k$$

$$
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\
a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn}
\end{pmatrix}
\begin{pmatrix}
m_1 \\ m_2 \\ m_3 \\ \cdots \\ \cdots \\ m_n
\end{pmatrix}
+
\begin{pmatrix}
b_1 \\ b_2 \\ b_3 \\ \cdots \\ \cdots \\ b_n
\end{pmatrix}
=
\begin{pmatrix}
c_1 \\ c_2 \\ c_3 \\ \cdots \\ \cdots \\ c_n
\end{pmatrix}
$$

where (A, B) is the key, A is an invertible $n \times n$ matrix with $\gcd(\det(A), N) = 1^2$, $M_i = (m_1, m_2, \ldots m_n)^T$, $B = (b_1, b_2, \ldots b_n)^T$, $C_i = (c_1, c_2, \ldots c_n)^T$. The decryption is performed by:

$$M_i \equiv A^{-1}(C_i - B) \,(\text{mod } N) \quad i = 1, 2, \ldots, k,$$

$$
\begin{pmatrix}
m_1 \\ m_2 \\ m_3 \\ \cdots \\ \cdots \\ m_n
\end{pmatrix}
=
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\
a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn}
\end{pmatrix}^{-1}
\begin{pmatrix}
c_1 - b_1 \\ c_2 - b_2 \\ c_3 - b_3 \\ \cdots \\ \cdots \\ c_n - b_n
\end{pmatrix} .
$$

---

[2] This requirement is to ensure that the matrix A is invertible.

(v) *Exponential Ciphers*

Pohlig and Hellman [PoH:76] invented the exponential cipher in 1976. Let p be a prime number and let M be the numerical representation of the plaintext, with each letter of the plaintext replaced with its two-digit representation (00–25). That is, $A = 00$, $B = 01$, ..., $Z = 25$.

M is divided into blocks M_i (these are equal size blocks of m letters where the block size is approximately the same number of digits as p). The number of letters m per block is chosen such that:

$$\underbrace{2525\ldots25}_{m \text{ times}} < p < \underbrace{2525\ldots25}_{m+1 \text{ times}}$$

For example, for the prime 8191 a block size of $m = 2$ letters is chosen since:

$$2525 < 8191 < 252525.$$

The secret encryption key is chosen to be an integer k such that $0 < k < p$ and $\gcd(k, p-1) = 1$. Then the encryption of the block M_i is defined by:

$$C_i = E_k(M_i) \equiv M_i{}^k (\text{mod } p).$$

The ciphertext C_i is an integer such that $0 < C_i < p$.

The decryption of C_i involves first determining the inverse k^{-1} of the key $k \,(\text{mod } p-1)$, i.e. we determine k^{-1} such that $k \cdot k^{-1} \equiv 1 \,(\text{mod } p-1)$. The secret key k was chosen so that $(k, p-1) = 1$, and this means that there are integers d and n such that $kd = 1 + n(p-1)$, and so k^{-1} is d and $kk^{-1} = 1 + n(p-1)$. Therefore,

$$D_{k^{-1}}(C_i) \equiv C_i{}^{k^{-1}} \equiv (M_i^k)^{k^{-1}} \equiv M_i{}^{1+n(p-1)} \equiv M_i(\text{mod } p).$$

The fact that $M_i^{1+n(p-1)} \equiv M_i(\text{mod } p)$ follows from Euler's Theorem and Fermat's Little Theorem which were discussed in Chap. 7. Euler's Theorem states that for two positive integers a and n with $\gcd(a, n) = 1$ then $a^{\phi(n)} \equiv 1(\text{mod } n)$.

Clearly, for a prime p we have that $\phi(p) = p - 1$. This allows us to deduce that:

$$M_i{}^{1+n(p-1)} \equiv M_i{}^1 M_i{}^{n(p-1)} \equiv M_i(M_i{}^{(p-1)})^n \equiv M_i(1)^n \equiv M_i(\text{mod } p).$$

(vi) *Data Encryption Standard (DES)*

DES is a popular cryptographic system [Nbs:77] used by governments and private companies around the world. It is based on a symmetric key algorithm and uses a shared secret key that is known only to the sender and receiver. It was designed by IBM and approved by the National Bureau of Standards (NBS[3]) in 1976. It is a block cipher and a message is split into 64-bit message blocks. The algorithm is employed in reverse to decrypt each ciphertext block.

---

[3] The NBS is now known as the National Institute of Standards and Technology (NIST).

Table 8.3 DES encryption

Step	Description
1.	Expansion of the 32-bit half block to 48 bits (by duplicating half of the bits)
2.	The 48-bit result is combined with a 48-bit subkey of the secret key using an XOR operation
3.	The 48-bit result is broken in to $8 * 6$ bits and passed through 8 substitution boxes to yield $8 * 4 = 32$ bits (This is the core part of the encryption algorithm)
4.	The 32-bit output is re-arranged according to a fixed permutation

Today, DES is considered to be insecure for many applications as its key size (56 bits) is viewed as being too small. The cipher has been broken in less than 24 h, and this has led to it being withdrawn as a standard and replaced by the Advanced Encryption Standard (AES). AES uses a larger key of 128 bits or 256 bits.

The DES algorithm uses the same secret 56-bit key for encryption and decryption. The key consists of 56 bits taken from a 64-bit key that includes 8 parity bits. The parity bits are at position 8, 16, ... , 64, and so every eighth bit of the 64-bit key is discarded leaving behind only the 56-bit key.

The algorithm is then applied to each 64-bit message block and the plaintext message block is converted into a 64-bit ciphertext block. An initial permutation is first applied to M to create M′, and M′ is divided into a 32-bit left half L_0 and a 32-bit right half R_0. There are then 16 iterations, with the iterations having a left half and a right half:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

The function f is a function that takes a 32-bit right half and a 48-bit round key K_i (each K_i contains a different subset of the 56-bit key) and produces a 32-bit output. Finally, the pre-ciphertext (R_{16}, L_{16}) is permuted to yield the final ciphertext C. The function f operates on half a message block and involves the steps in Table 8.3.

The decryption of the ciphertext is similar to the encryption and it involves running the algorithm in reverse.

DES has been implemented on a microchip. However, it has been superseded in recent years by AES due to security concerns with its small 56-bit key size.

8.5 Public Key Systems

A public key cryptosystem is an asymmetric key system where there is a separate key e_k for encryption and d_k decryption with $e_k \neq d_k$. Martin Hellman and Whitfield
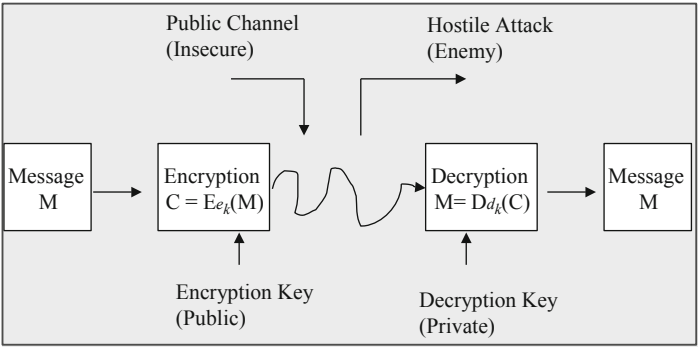
Fig. 8.7 Public key cryptosystem

Table 8.4 Public key
encryption system

Item	Description
1.	It uses the concept of a key pair (e_k, d_k)
2.	One half of the pair can encrypt messages and the other half can decrypt messages
3.	One key is private and one key is public
4.	The private key is kept secret and the public key is published (but associated with trading partner)
5.	The key pair is associated with exactly one trading partner

Diffie invented it in 1976 (Fig. 8.7). The fact that a person is able to encrypt a message does not mean that the person has sufficient information to decrypt messages.

The public key cryptosystem is based on the items in Table 8.4.

The advantages and disadvantages of public key cryptosystems are included in Table 8.5.

The implementation of public-key cryptosystems is based on *trapdoor one-way functions*. A function $f : X \rightarrow Y$ is a trapdoor one-way function if

- f is easy to computer
- f^{-1} is difficult to compute
- f^{-1} is easy to compute if a trapdoor (secret information associated with the function) becomes available.

A function satisfying just the first two conditions above is termed a *one-way function*.

Examples of Trapdoor and One-Way Functions

(i) The function $f : pq \rightarrow n$ (where p and q are primes) is a one-way function since it is easy to compute. However, the inverse function f^{-1} is difficult to compute problem for large n since there is no efficient algorithm to factorise a large integer into its prime factors (*integer factorisation problem*).

(ii) The function $f_{g,N} : x \rightarrow g^x (\mathrm{mod}\ N)$ is a one-way function since it is easy to compute. However, the inverse function f^{-1} is difficult to compute as there is

Table 8.5 Advantages and
disadvantages of public key
cryptosystems

Advantages	Disadvantages
Only the private key needs to be kept secret	Public keys must be authenticated
The distribution of keys for encryption is convenient as everyone publishes their public key and the private key is kept private	It is slow and uses more computer resources
It provides message authentication as it allows the use of digital signatures (which enables the recipient to verify that the message is really from the particular sender)	Security Compromise is possible (if private key compromised)
The sender encodes with the private key that is known only to sender. The receiver decodes with the public key and therefore knows that the message is from the sender	Loss of private key may be irreparable (unable to decrypt messages)
Detection of tampering (digital signatures enable the receiver to detect whether message was altered in transit)	
Provides for non-repudiation	

no efficient method to determine x from the knowledge of $g^x (\text{mod N})$ and g and
N (*the discrete logarithm problem*).

(iii) The function $f_{k,N} : x \to x^k (\text{mod N})$ (where N $= pq$ and p and q are primes) and
$kk' \equiv 1 \bmod \varphi(n)$ is a trapdoor function. It is easy to compute but the inverse of
f (the kth root modulo N) is difficult to compute. However, if the trapdoor k' is
given then f can easily be inverted as $(x^k)^{k'} \equiv x (\text{mod N})$.

8.5.1   RSA Public Key Cryptosystem

Rivest, Shamir and Adleman proposed a practical public key cryptosystem (RSA)
based on primality testing and integer factorisation in the late 1970s. The RSA algo-
rithm was filed as a patent (Patent No. 4,405,829) at the US Patent Office in December
1977. The RSA public key cryptosystem is based on the following assumptions:

- It is straightforward to find two large prime numbers.
- The integer factorisation problem is infeasible for large numbers.

The algorithm is based on mod-n arithmetic where n is a product of two large prime
numbers.

The encryption of a plaintext message M to produce the ciphertext C is given by:

$$C \equiv M^e (\text{mod } n)$$

where e is the public encryption key, M is the plaintext, C is the ciphertext and n
is the product of two large primes p and q. Both e and n are made public, and e is

Table 8.6 Steps for A to send secure message and signature to B

Step	Description
1.	A uses B's public key to encrypt the message
2.	A uses its private key to encrypt its signature
3.	A sends the message and signature to B
4.	B uses A's public key to decrypt A's signature
5.	B uses its private key to decrypt A's message

chosen such that $1 < e < \phi(n)$, where $\phi(n)$ is the number of positive integers that are relatively prime to n.

The ciphertext C is decrypted by

$$M \equiv C^d (\mathrm{mod}\ n)$$

where d is the private decryption key that is known only to the receiver, and $ed \equiv 1 \pmod{\phi(n)}$ and d and $\phi(n)$ are kept private.

The calculation of $\phi(n)$ is easy if both p and q are known, as it is given by $\phi(n) = (p-1)(q-1)$. However, its calculation for large n is infeasible if p and q are unknown,

$$ed \equiv 1 (\mathrm{mod}\ \phi(n)),$$
$$\Rightarrow ed = 1 + k\ \phi(n) \text{for some } k \in \mathbb{Z}$$

We discussed Euler's Theorem in Chap. 7, and this result states that if a and n are positive integers with $\gcd(a, n) = 1$ then $a^{\phi(n)} \equiv 1 (\mathrm{mod}\ n)$. Therefore, $M^{\phi(n)} \equiv 1 (\mathrm{mod}\ n)$ and so $M^{k\phi(n)} \equiv 1 (\mathrm{mod}\ n)$. The decryption of the ciphertext is given by:

$$C^d (\mathrm{mod}\ n) \equiv M^{ed} (\mathrm{mod}\ n)$$

$$\equiv M^{1+k\phi(n)} (\mathrm{mod}\ n)$$

$$\equiv M^1 M^{k\phi(n)} (\mathrm{mod}\ n)$$

$$\equiv M.1 (\mathrm{mod}\ n)$$

$$\equiv M (\mathrm{mod}\ n).$$

8.5.2 Digital Signatures

The RSA public-key cryptography may also be employed to obtain digital signatures. Suppose *A* wishes to send a secure message to *B* as well as a digital signature. This involves signature generation using the private key, and signature verification using the public key. The steps involved are provided in Table 8.6.

The National Institute of Standards and Technology (NIST) proposed an algorithm for digital signatures in 1991. The algorithm is known as the Digital Signature Algorithm (DSA) and later became the Digital Signature Standard (DSS).

8.6   Review Questions

1. Discuss the early ciphers developed by Julius Caesar and Augustus. How effective were they at that period in history, and what are their weaknesses today?
2. Describe how the team at Bletchley Park cracked the German Enigma codes.
3. Explain the differences between a public key cryptosystem and a private key cryptosystem.
4. What are the advantages and disadvantages of private (symmetric) key cryptosystems?
5. Describe the various types of symmetric key systems.
6. What are the advantages and disadvantages of public key cryptosystems?
7. Describe public key cryptosystems including the RSA public key cryptosystem.
8. Describe how digital signatures may be generated.

8.7   Summary

This chapter provided a brief introduction to cryptography, which is the study of mathematical techniques that provide secrecy in the transmission of messages between computers. It was originally employed to protect communication between individuals, and today it is employed to solve security problems such as privacy and authentication over a communications channel.

It involves enciphering and deciphering messages, and theoretical results from number theory are employed to convert the original messages (or plaintext) into ciphertext that is then transmitted over a secure channel to the intended recipient. The ciphertext is meaningless to anyone other than the intended recipient, and the received ciphertext is then decrypted to allow the recipient to read the message.

A *public key cryptosystem* is an asymmetric cryptosystem. It has two different encryption and decryption keys, and the fact that a person has knowledge on how to encrypt messages does not mean that the person has sufficient information to decrypt messages.

In a *secret key cryptosystem* the same key is used for both encryption and decryption. Anyone who has knowledge on how to encrypt messages has sufficient knowledge to decrypt messages.

Chapter 9
Coding Theory

Key Topics
  Groups, Rings and Fields
  Block Codes
  Error Detection and Correction
  Generation Matrix
  Hamming Codes

9.1 Introduction

Coding theory is a practical branch of mathematics concerned with the reliable transmission of information over communication channels. It allows errors to be detected and corrected, which is essential when messages are transmitted through a noisy communication channel. The channel could be a telephone line, radio link or satellite link, and coding theory is applicable to mobile communications and satellite communications. It is also applicable to storing information on storage systems such as the compact disc.

It includes theory and practical algorithms for error detection and correction, and this is essential in modern communication systems that require reliable and efficient transmission of information.

An error-correcting code encodes the data by adding a certain amount of redundancy to the message. This enables the original message to be recovered if a small number of errors have occurred. The extra symbols added are also subject to errors, as accurate transmission cannot be guaranteed in a noisy channel.

The basic structure of a digital communication system is shown in Fig. 9.1. It includes transmission tasks such as source encoding, channel encoding and modulation; and receiving tasks such as demodulation, channel decoding and source decoding.

The modulator generates the signal that is used to transmit the sequence of symbols b across the channel. The transmitted signal may be altered due to the fact that there
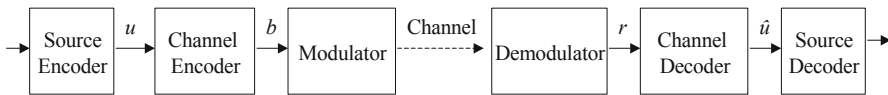
Fig. 9.1 Basic digital communication

is noise in the channel, and the signal received is demodulated to yield the sequence of received symbols r.

The received symbol sequence r may differ from the transmitted symbol sequence b due to the noise in the channel, and therefore a channel code is employed to enable errors to be detected and corrected. The channel encoder introduces redundancy into the information sequence u, and the channel decoder uses the redundancy for error detection and correction. This enables the transmitted symbol sequence $\hat{u}$ to be estimated.

Shannon [Sha:48] showed that it is theoretically possible to produce an information transmission system with an error probability as small as required provided that the information rate is smaller than the channel capacity.

Coding theory uses several results from pure mathematics, and so first the mathematical foundations of coding theory are considered.

9.2 Mathematical Foundations

Coding theory uses results from modern algebra, and algebraic structures such as groups, rings, fields and vector spaces are employed to provide a solid foundation for the discipline.

A *group* is a non-empty set with a single binary operation whereas *rings* and *fields* are algebraic structures with two binary operations satisfying various laws. A *vector space* consists of vectors over a field.

Each of these abstract mathematical structures is discussed below, and concrete examples are presented.

9.2.1 Groups

A non-empty set G together with a binary operation '*' is called a *group* if for all elements a, b, $c \in$ G the following properties hold:

1. $a * b \in$ G (Closure property)
2. $a * (b * c) = (a * b) * c$ (Associative property)
3. $\exists e \in$ G such that : $a * e = e * a = a (\forall a \in$ G) (Identity Element)
4. For every $a \in$ G, $\exists a^{-1} \in$ G, such that: $a * a^{-1} = a^{-1} * a = e$ (Inverse Element)

The identity element is unique, and the inverse a^{-1} of an element a is unique. A *commutative group* has the additional property that for all $a, b \in G$,

$$a * b = b * a.$$

The order of a group G is the number of elements in G, and is denoted by $o(G)$. If the order of G is finite then G is said to be a finite group. A *semi-group* (M, *) is a set with a binary operation '*' such that the closure and associativity properties hold. A *monoid* is a semi-group with an identity element.

Example 9.1 (Groups)

1. The set of integers under addition forms an infinite group in which 0 is the identity element.
2. The set of integer 2×2 matrices over the real numbers under matrix addition, where the identity element is $\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$.
3. The set of integers under multiplication forms an infinite monoid with 1 as the identity element.

A *cyclic group* is a group where all elements $g \in G$ are obtained from the powers a^i of one element $a \in G$, with $a^0 = e$. The element 'a' is termed the generator of the cyclic group G. A finite cyclic group is with n elements is of the form $\{a^0, a^1, a^2, \ldots, a^{n-1}\}$.

A non-empty subset H of a group G is said to be a *subgroup* of G if for all $a, b \in H$ then $a^*b \in H$, and for any $a \in H$ then $a^{-1} \in H$.

Lagrange's theorem states the relationship between the order of a subgroup H of G, and the order of G. The theorem states that if G is a finite group, and H is a subgroup of G, then $o(H)$ is a divisor of $o(G)$.

9.2.2   Rings

A *ring* is a non-empty set R together with two binary operation '+' and '×' where (R, +) is a commutative group; (R, ×) is a semi-group; and the left and right distributive laws hold. Specifically, for all elements $a, b, c \in R$ the following properties hold:

1. $a + b \in R$ (Closure property)
2. $a + (b + c) = (a + b) + c$ (Associative property)
3. $\exists 0 \in R$ such that $\forall a \in R : a + 0 = 0 + a = a$ (Identity Element)
4. $\forall a \in R : \exists (-a) \in R : a + (-a) = (-a) + a = 0$ (Inverse Element)
5. $a + b = b + a$ (Commutativity)
6. $a \times b \in R$ (Closure property)
7. $a \times (b \times c) = (a \times b) \times c$ (Associative property)
8. $a(b + c) = a \times b + a \times c$ (Distributive Law)
9. $(b + c) \times a = b \times a + c \times a$ (Distributive Law)

The element 0 is the identity element under addition, and the additive inverse of an element a is given by $-a$. If a ring (R, $\times$, $+$) has a multiplicative identity 1 where $a \times 1 = 1 \times a = a$ for all $a \in$ R then R is termed a ring with a unit element. If $a \times b = b \times a$ for all $a, b \in$ R then R is termed a *commutative ring*.

An element $a \neq 0$ in a ring R is said to be a *zero divisor* if there exists $b \in$ R, with $b \neq 0$ such that $ab = 0$. A commutative ring is an *integral domain* if it has no zero divisors. A ring is said to be a *division ring* if its non-zero elements form a group under multiplication.

Example 9.2 (Rings)

1. The set of integers $(\mathbb{Z}, +, \times)$ forms an infinite commutative ring with multiplicative unit element 1. Further, since it has no zero divisors it is an integral domain.
2. The set of integers mod 4 (i.e. $\mathbb{Z}_4$[1] where addition and multiplication is performed modulo 4) is a finite commutative ring with unit element $[1]_4$. Its elements are $\{[0]_4, [1]_4, [2]_4, [3]_4\}$. It has zero divisors since $[2]_4[2]_4 = [0]_4$ and so it is not an integral domain.
3. The quaternions (discussed in a later chapter) are an example of a non-commutative ring (they form a division ring).
4. The set of integers mod 5 (i.e. $\mathbb{Z}_5$ where addition and multiplication is performed modulo 5) is a finite commutative division ring[2].

9.2.3   Fields

A *field* is a non-empty set F together with two binary operation '$+$' and '$\times$' where (F, $+$) is a commutative group; (F\{0\}, $\times$) is a commutative group; and the distributive properties hold. The properties of a field are:

1. $a + b \in$ F (Closure property)
2. $a + (b + c) = (a + b) + c$ (Associative property)
3. $\exists 0 \in$ F such that $\forall a \in$ F: $a + 0 = 0 + a = a$ (Identity Element)
4. $\forall a \in$ F $: \exists(-a) \in$ F: $a + (-a) = (-a) + a = 0$ (Inverse Element)
5. $a + b = b + a$ (Commutativity)
6. $a \times b \in$ F (Closure property)
7. $a \times (b \times c) = (a \times b) \times c$ (Associative property)
8. $\exists 1 \in$ F such that $\forall a \in$ F: $a \times 1 = 1 \times a = a$ (Identity Element)
9. $\forall a \in$ F $\setminus\{0\}, \exists a^{-1} \in$ F: $a \times a^{-1} = a^{-1} \times a = 1$ (Inverse Element)
10. $a \times b = b \times a$ (Commutativity)
11. $a \times (b + c) = a \times b + a \times c$ (Distributive Law)
12. $(b + c) \times a = b \times a + c \times a$ (Distributive Law)

---

[1] Recall from Chap. 7 that $\mathbb{Z}/n\mathbb{Z} = \mathbb{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\} = \{[0]_n, [1]_n, \ldots, [n-1]_n\}$.

[2] A finite division ring is actually a field (i.e. it is commutative under multiplication).

The following are examples of fields:

Example 9.3 (Fields)

1. The set of rational numbers $(\mathbb{Q}, +, \times)$ forms an infinite commutative field. The additive identity is 0, and the multiplicative identity is 1.
2. The set of real numbers $(\mathbb{R}, +, \times)$ forms an infinite commutative field. The additive identity is 0, and the multiplicative identity is 1.
3. The set of complex numbers $(\mathbb{C}, +, \times)$ forms an infinite commutative field. The additive identity is 0, and the multiplicative is 1.
4. The set of integers mod 7 (i.e. $\mathbb{Z}_7$ where addition and multiplication is performed mod 7) is a finite field.
5. The set of integers mod p where p is a prime (i.e. $\mathbb{Z}_p$ where addition and multiplication is performed mod p) is a finite field with p elements. The additive identity is [0] and the multiplicative identity is [1].

A field is a commutative division ring but not every division ring is a field. For example, the quaternions (discussed in a later chapter) are an example of a division ring, which is not a field. However, a finite division ring is a field.

If the number of elements in the field F is finite then F is called a finite field, and F is written as F_q where q is the number of elements in F. In fact, every finite field has $q = p^k$ elements for some prime p, and some $k \in \mathbb{N}$.

9.2.4 Vector Spaces

A non-empty set V is said to be a *vector space* over a field F if V is a commutative group under vector addition $+$ and if for every $\alpha \in F, v \in V$ there is an element αv in V such that the following properties hold for $v, w \in V$ and $\alpha, \beta \in F$:

1. $u + v \in V$
2. $u + (v + w) = (u + v) + w$
3. $\exists 0 \in V$ such that $\forall v \in V : v + 0 = 0 + v = v$
4. $\forall v \in V : \exists(-v) \in V : v + (-v) = (-v) + v = 0$
5. $v + w = w + v$
6. $\alpha(v + w) = \alpha v + \alpha w \qquad \alpha \in F, v, w, \in V$
7. $(\alpha + \beta)v = \alpha v + \beta v \qquad \alpha, \beta \in F, v \in V$
8. $\alpha(\beta v) = (\alpha \beta)v \qquad \alpha, \beta \in F, v \in V$
9. $1v = v$

The elements in V are referred to as *vectors* and the elements in F are referred to as *scalars*. The element 1 refers to the identity element of the field F under multiplication.

The representation of codewords, which is discussed later, is by n-dimensional vectors over the finite field F_q. A codeword vector v is represented as the n-tuple:

$$v = (a_0, a_1, \ldots, a_{n-1}),$$

where each $a_i \in F_q$. The set of all n-dimensional vectors is the n-dimensional vector space $\mathbf{F}^n{}_q$ with q^n elements. The addition of two vectors v and w, where $v = (a_0, a_1, \ldots a_{n-1})$ and $w = (b_0, b_1, \ldots b_{n-1})$ is given by:

$$v + w = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1}).$$

The scalar multiplication of a vector $v = (a_0, a_1, \ldots a_{n-1}) \in \mathbf{F}^n{}_q$ by a scalar $\beta \in F_q$ is given by:

$$\beta v = (\beta a_0, \beta a_1, \ldots, \beta a_{n-1}).$$

The set $\mathbf{F}^n{}_q$ is called the vector space over the finite field F_q, if the vector space properties above hold. A finite set of vectors $v_1, v_2, \ldots v_k$ is said to be *linearly independent* if

$$\beta_1 v_1 + \beta_2 v_2 + \cdots + \beta_k v_k = 0 \Rightarrow \beta_1 = \beta_2 \ldots \beta_k = 0.$$

Otherwise, the set of vectors $v_1, v_2, \ldots v_k$ is said to be *linearly dependent*.

A non-empty subset W of a vector space V ($W \subseteq V$) is said to be a *subspace* of V, if W forms a vector space over F under the operations of V. This is equivalent to W being closed under vector addition and scalar multiplication: i.e. $w_1, w_2 \in W$, α, $\beta \in F$ then $\alpha w_1 + \beta w_2 \in W$.

The *dimension* (dim W) of a subspace $W \subseteq V$ is k if there are k linearly independent vectors in W but every $k + 1$ vectors are linearly dependent. A subset of a vector space is a *basis* for V if it consists of linearly independent vectors, and its linear span is V (i.e. the basis generates V). We shall employ the basis of the vector space of codewords to create the generator matrix to simplify the encoding of the information words. The linear span of a set of vectors $v_1, v_2, \ldots, v_k$ is defined as $\beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_k v_k$.

Example 9.4 (Vector Spaces)

1. The real coordinate space $\mathbb{R}^n$ forms an n-dimensional vector space over $\mathbb{R}$. The elements of $\mathbb{R}^n$ are the set of all n tuples of elements of $\mathbb{R}$, where an element x in $\mathbb{R}^n$ is written as:

$$x = (x_1, x_2, \ldots, x_n),$$

   where each $x_i \in \mathbb{R}$ and vector addition and scalar multiplication are given by:

$$ax = (ax_1, ax_2, \ldots, ax_n),$$
$$x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n).$$

2. The set of $m \times n$ matrices over the real numbers forms a vector space, with vector addition given by matrix addition, and the multiplication of a matrix by a scalar given by the multiplication of each entry in the matrix by the scalar.

9.3 Simple Channel Code

This section considers an example to illustrate the concept of an error-correcting code. This example code is able to correct a single transmitted error only.

The transmission of binary information over a noisy channel leads to differences between the transmitted sequence and the received sequence. These differences are illustrated by underlining the relevant digits. For example:

$$\text{Sent} \quad 00\underline{10}1110$$

$$\text{Received} \quad 00\underline{00}0110$$

It is assumed that initially the transmission is done without channel codes:

$$00101110 \xrightarrow{\text{Channel}} 00000110$$

Next, the use of an encoder is considered and a triple repetition-encoding scheme is employed. That is, the binary symbol 0 is represented by the codeword 000, and the binary symbol 1 is represented by the codeword 111.

$$00101110 \rightarrow \boxed{\text{Encoder}} \rightarrow 000000111000111111111000$$

In other words, if the symbol 0 is to be transmitted then the encoder emits the codeword 000, and similarly the encoder emits 111 if the symbol 1 is to be transmitted. Assuming that on average one symbol in four is incorrectly transmitted then transmission with binary triple repetition may result in a received sequence such as:

$$000000111000111111111000 \rightarrow \boxed{\text{Channel}} \rightarrow 0\underline{1}00000110\underline{1}0111\underline{0}1\underline{0}11110\underline{1}0$$

The decoder tries to estimate the original sequence by using a *majority decision* on each 3-bit word. Any 3-bit word that contains more zeros than ones is decoded to 0, and similarly if it contains more ones than zero it is decoded to 1. The decoding algorithm yields:

$$0\underline{1}00000110\underline{1}0111\underline{0}1\underline{0}11110\underline{1}0 \rightarrow \boxed{\text{Decoder}} \rightarrow 00101\underline{0}10$$

In this example, the binary triple repetition code is able to correct a single error within a codeword (as the majority decision is two to one). This helps to reduce the number of errors transmitted compared to unprotected transmission. In the first case where an encoder is not employed there are two errors, whereas there is just one error when the encoder is used.

However, there are disadvantages with this approach in that the transmission bandwidth has been significantly reduced. It now takes three times as long to transmit an information symbol with the triple replication code than with standard transmission. Therefore, it is desirable to find more efficient coding schemes.

9.4   Block Codes

There were two codewords employed in the simple example above: namely 000 and 111. This is an example of a (n, k) code where the codewords are of length $n = 3$, and the information words are of length $k = 1$ (as we were just encoding a single symbol 0 or 1). This is an example of a (3, 1) block code, and the objective of this section is to generalise the simple coding scheme to more efficient and powerful channel codes.

The fundamentals of the q-nary (n, k) block codes (where q is the number of elements in the finite field F_q) involve converting an information block of length k to a codeword of length n. Consider an information sequence $u_0, u_1, u_2, \ldots$ of discrete information symbols where $u_i \in \{0, 1, \ldots q - 1\} = F_q$. The normal class of channel codes is when we are dealing with binary codes: i.e. $q = 2$. The information sequence is then grouped into blocks of length k as follows:

$$\underbrace{u_0 u_1 u_2 \ldots u_{k-1}} \quad \underbrace{u_k u_{k+1} u_{k+2} \ldots u_{2k-1}} \quad \underbrace{u_{2k} u_{2k+1} u_{2k+2} \ldots u_{3k-1}} \ldots \ldots$$

Each block is of length k (i.e. the information words are of length k), and it is then encoded separately into codewords of length n. For example, the information word $u_0 u_1 u_2 \ldots u_{k-1}$ is uniquely mapped to a codeword $b_0 b_1 b_2 \ldots b_{n-1}$ (where $b_i \in F_q$):

$$(u_0 u_1 u_2 \ldots u_{k-1}) \rightarrow \boxed{\text{Encoder}} \rightarrow (b_0 b_1 b_2 \ldots b_{n-1})$$

These codewords are then transmitted across the communication channel and the received words are then decoded. The received word $r = (r_0 r_1 r_2 \ldots r_{n-1})$ is decoded into the information word $\hat{u} = (\hat{u}_0 \hat{u}_1 \hat{u}_2 \ldots \hat{u}_{k-1})$.

$$(r_0 r_1 r_2 \ldots r_{n-1}) \rightarrow \boxed{\text{Decoder}} \rightarrow (\hat{u}_0 \hat{u}_1 \hat{u}_2 \ldots \hat{u}_{k-1})$$

The decoding is done in two steps with the received n-block word r first decoded to an n-block codeword, which is then decoded into the k-block information word $\hat{u}$. The encoding, transmission and decoding of an (n, k) block is summarised in Fig. 9.2.

A lookup table may be employed for the encoding to determine the codeword b for each information word u. However, the size of the table grows exponentially with increasing information word length k, and so this is inefficient due to the large memory size required. We shall discuss later how a generator matrix provides an efficient encoding and decoding mechanism.

Notes

1. The codeword is of length n.
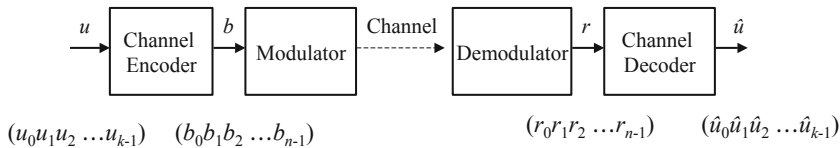2. The information word is of length k.

$$u \quad \fbox{Channel Encoder} \quad b \quad \fbox{Modulator} \quad \text{Channel} \quad \fbox{Demodulator} \quad r \quad \fbox{Channel Decoder} \quad \hat{u}$$

$(u_0 u_1 u_2 \ldots u_{k-1}) \quad (b_0 b_1 b_2 \ldots b_{n-1}) \qquad\qquad (r_0 r_1 r_2 \ldots r_{n-1}) \quad (\hat{u}_0 \hat{u}_1 \hat{u}_2 \ldots \hat{u}_{k-1})$

Fig. 9.2 Encoding and decoding of an (n, k) block

3. The codeword length n is larger than the information word length k.
4. A block (n, k) code is a code in which all codewords are of length n and all information words are of length k.
5. The number of possible information words is given by $M = q^k$ (where each information symbol can take one of q possible values and the length of the information word is k).
6. The code rate R in which information is transmitted across the channel is given by:

$$R = \frac{k}{n}.$$

7. The weight of a codeword is $b = (b_0 b_1 b_2 \ldots b_{n-1})$ is given by the number of non-zero components of b. That is,

$$wt(b) = |\{i : b_i \neq 0, 0 \leq i < n\}|.$$

8. The distance between two codewords $b = (b_0 b_1 b_2 \ldots b_{n-1})$ and $b' = (b_0' b_1' b_2' \ldots b_{n-1}')$ measures how close the codewords b and b' are to each other. It is given by the Hamming distance:

$$\text{dist}(b, b') = \left|\{i : b_i \neq b'_i, 0 \leq i < n\}\right|.$$

9. The minimum Hamming distance for a code **B** consisting of M codewords $b_1, \ldots, b_M$ is given by:

$$d = \min\{\text{dist}(b, b') : \text{where } b \neq b' \text{and } b, b' \in \mathbf{B}\}.$$

10. The (n, k) block code $\mathbf{B} = \{b_1, \ldots, b_M\}$ with M $(= q^k)$ codewords of length n and minimum Hamming distance d is denoted by $\mathbf{B}(n, k, d)$.

9.4.1   Error Detection and Correction

The minimum Hamming distance offers a way to assess the error-detection and error-correction capability of a channel code. Consider two codewords b and b' of an (n, k) block code $\mathbf{B}(n, k, d)$.

Fig. 9.3 Error-correcting
capability sphere
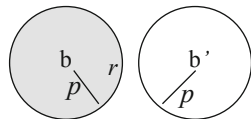
Then, the distance between these two codewords is greater than or equal to the minimum Hamming distance d, and so errors can be detected as long as the erroneously received word is not equal to a codeword different from the transmitted codeword.

That is, the *error-detection capability* is guaranteed as long as the number of errors is less than the minimum Hamming distance d, and so the number of detectable errors is $d - 1$.

Any two codewords are of distance at least d and so if the number of errors is less than $d/2$ then the received word can be properly decoded to the codeword b. That is, the *error-correction capability* is given by:

$$E_{\text{cor}} = \frac{d - 1}{2}.$$

An error-correcting sphere may be employed to illustrate the error correction of a received word to the correct codeword b. This may be done when all received words are within the error-correcting sphere with radius p $(< d/2)$ (Fig. 9.3).

If the received word r is different from b in less than $d/2$ positions, then it is decoded to b as it is less than $d/2$ positions from the next closest codeword b'. That is, b is the closest codeword to the received word r (provided that the error-correcting radius is less than $d/2$).

9.5   Linear Block Codes

Linear block codes have nice algebraic properties, and the codewords in a linear block code are considered to be vectors in the finite vector space $\mathbf{F}^n_q$. The representation of codewords by vectors allows the nice algebraic properties of vector spaces to be used, and this simplifies the encoding of information words as a generator matrix may be employed to create the codewords.

An (n, k) block code $\mathbf{B}(n, k, d)$ with minimum Hamming distance d over the finite field F_q is called *linear* if $\mathbf{B}(n, k, d)$ is a subspace of the vector space $\mathbf{F}^n_q$ of dimension k. The number of codewords is then given by:

$$M = q^k.$$

The rate of information (R) through the channel is given by:

$$R = \frac{k}{n}.$$

Fig. 9.4 Generator matrix

$$G = \begin{pmatrix} g_{0,} \\ g_{1,} \\ g_{2,} \\ \cdots \\ \cdots \\ \cdots \\ g_{k-1} \end{pmatrix} = \begin{pmatrix} g_{0,0} & g_{0,1} & g_{0,2} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & g_{1,2} & \cdots & g_{1,n-1} \\ g_{2,0} & g_{2,1} & g_{2,2} & \cdots & g_{2,n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ g_{k-1,0} & g_{k-1,1} & g_{k-1,2} & \cdots & g_{k-1,n-1} \end{pmatrix}$$

Clearly, since $\mathbf{B}(n, k, d)$ is a subspace of $\mathbf{F}^n{}_q$ any linear combination of the codewords (vectors) will be a codeword. That is, for the codewords $b_1, b_2, \ldots, b_r$ we have that:

$$\alpha_1 b_1 + \alpha_2 b_2 + \cdots + \alpha_r b_r \in B(n, k, d),$$

where $\alpha_1, \alpha_2, \ldots, \alpha_r \in F_q$ and $b_1, b_2, \ldots, b_r \in \mathbf{B}(n, k, d)$.

Clearly, the n-dimensional zero row vector $(0, 0, \ldots, 0)$ is always a codeword, and so $(0, 0, \ldots, 0) \in \mathbf{B}(n, k, d)$. The minimum Hamming distance of a linear block code $\mathbf{B}(n, k, d)$ is equal to the minimum weight of the non-zero codewords: That is,

$$d = \min\{\text{dist } (b, b')\} = \min_{\forall b \neq 0} \text{wt}(b).$$
$${\scriptstyle \forall b \neq b'}$$

In summary, an (n, k) linear block code $\mathbf{B}(n, k, d)$ is:

1. A subspace of $\mathbf{F}^n{}_q$.
2. The number of codewords is $M = q^k$.
3. The minimum Hamming distance d is the minimum weigh of the non-zero codewords.

The encoding of a specific k-dimensional information word $u = (u_0, u_1, \ldots, u_{k-1})$ to a n-dimensional codeword $b = (b_0, b_1, \ldots, b_{n-1})$ may be done efficiently with a generator matrix. First, a basis $\{g_0, g_1, \ldots, g_{k-1}\}$ of the k-dimensional subspace spanned by the linear block code is chosen, and this consists of k linearly independent n-dimensional vectors. Each basis element g_i (where $0 \leq i \leq k - 1$) is a n-dimensional vector:

$$g_i = (g_{i,0}, g_{i,1}, \ldots, g_{i,n-1}).$$

The corresponding codeword $b = (b_0, b_1, \ldots, b_{n-1})$ is then a linear combination of the information word with the basis elements. That is,

$$b = u_0 g_0 + u_1 g_1 + \cdots + u_{k-1} g_{k-1},$$

where each information symbol $u_i \in F_q$. The generator matrix G is then constructed from the k linearly independent basis vectors as shown in Fig. 9.4.

The encoding of the k-dimensional information word u to the n-dimensional codeword b involves matrix multiplication (Fig. 9.5).

$$(u_0, u_1, ..., u_{k-1}) \begin{pmatrix} g_{0,0} & g_{0,1} & g_{0,2} & .... & g_{0,n-1} \\ g_{1,0} & g_{1,1} & g_{1,2} & .... & g_{1,n-1} \\ g_{2,0} & g_{2,1} & g_{2,2} & .... & g_{2,n-1} \\ ... & .... & .... & .... & .... \\ ... & .... & .... & .... & .... \\ ... & .... & .... & .... & .... \\ g_{k-1,0} & g_{k-1,1} & g_{k-1,2} & .... & g_{k-1,n-1} \end{pmatrix} = (b_0, b_1, ..., b_{n-1})$$

Fig. 9.5 Generation of codewords

Fig. 9.6 Identity matrix
$(k \times k)$

$$I_k = \begin{pmatrix} 1 & 0 & 0 & .... & 0 \\ 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & .... & 0 \\ ... & .... & .... & .... & .... \\ ... & .... & .... & .... & .... \\ ... & .... & .... & .... & .... \\ 0 & 0 & 0 & .... & 1 \end{pmatrix}$$

This may also be written as:

$$b = uG.$$

Clearly, all $M = q^k$ codewords $b \in \mathbf{B}(n, k, d)$ can be generated according to this rule, and so the matrix G is called the generator matrix. The generator matrix defines the linear block code $\mathbf{B}(n, k, d)$.

There is an equivalent $k \times n$ generator matrix for $\mathbf{B}(n, k, d)$ defined as:

$$G = I_k \,|A_{k,n-k},$$

where I_k is the $k \times k$ identity matrix defined as shown in Fig. 9.6.

The encoding of the information word u yields the codeword b such that the first k symbols b_i of b are the same as the information symbols u_i $0 \le i \le k$,

$$b = uG = (u \,|uA_{k,n-k}).$$

The remaining $m = n - k$ symbols are generated from $uA_{k,n-k}$ and the last m symbols are the m parity-check symbols. These are attached to the information vector u for the purpose of error detection and correction.

9.5.1   Parity-Check Matrix

The linear block code $\mathbf{B}(n, k, d)$ with generator matrix $G = (I_k, | A_{k,n-k})$ may be defined equivalently by the $(n - k) \times n$ parity-check matrix H, where this matrix is

Fig. 9.7 Hamming code
B(7, 4, 3) generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

defined as:

$$H = (-A^T_{k,n-k} \,|\, I_{n-k}).$$

The generator matrix G and the parity-check matrix H are orthogonal: i.e.

$$HG^T = 0_{n-k,k}.$$

The parity-check orthogonality property holds if and only if the vector belongs to the linear block code. That is, for each code vector in $b \in \mathbf{B}(n, k, d)$ we have

$$Hb^T = 0_{n-k,1}$$

and vice verse whenever the property holds for a vector r, then r is a valid codeword in $\mathbf{B}(n, k, d)$. We present an example of a parity-check matrix in Example 9.5 below.

9.5.2 Binary Hamming Code

The Hamming code is a linear code that has been employed in dynamic random access memory to detect and correct deteriorated data in memory. The generator matrix for the $\mathbf{B}(7, 4, 3)$ binary Hamming code is given in Fig. 9.7.

The information words are of length $k = 4$ and the codewords are of length $n = 7$. For example, it can be verified by matrix multiplication that the information word (0, 0, 1, 1) is encoded into the codeword (0, 0, 1, 1, 0, 0, 1).

That is, three parity bits 001 have been added to the information word (0, 0, 1, 1) to yield the codeword (0, 0, 1, 1, 0, 0, 1).

The minimum Hamming distance is $d = 3$, and the Hamming code can detect up to two errors, and it can correct one error.

Example 9.5 (Parity-Check Matrix—Hamming Code) The objective of this example is to construct the parity-check matrix of the binary Hamming code (7, 4, 3), and to show an example of the parity-check orthogonality property.

First, we construct the parity-check matrix H which is given by $H = (-A^T_{k,n-k} \,|\, I_{n-k})$ or in other words $H = (-A^T_{4,3} \,|\, I_3)$. We first note that

$$A_{4,3} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad A^T_{4,3} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}.$$

Therefore, H is given by:

$$H = \begin{pmatrix} 0 & -1 & -1 & -1 & 1 & 0 & 0 \\ -1 & 0 & -1 & -1 & 0 & 1 & 0 \\ -1 & -1 & 0 & -1 & 0 & 0 & 1 \end{pmatrix}.$$

We noted that the encoding of the information word $u = (0011)$ yields the codeword $b = (0011001)$. Therefore, the calculation of Hb^{T} yields (recalling that addition is modulo 2):

$$Hb^{\mathrm{T}} = \begin{pmatrix} 0 & -1 & -1 & -1 & 1 & 0 & 0 \\ -1 & 0 & -1 & -1 & 0 & 1 & 0 \\ -1 & -1 & 0 & -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

9.5.3   Binary Parity-Check Code

The binary parity-check code is a linear block code over the finite field F_2. The code takes a k-dimensional information word $u = (u_0, u_1, \ldots u_{k-1})$ and generates the codeword $b = (b_0, b_1, \ldots, b_{k-1}, b_k)$ where $u_i = b_i$ ($0 \le i \le k-1$) and b_k is the parity bit chosen so that the resulting codeword is of even parity. That is,

$$b_k = u_0 + u_1 + \cdots + u_{k-1} = \sum_{i=0}^{k-1} u_i.$$

9.6   Miscellaneous Codes in Use

There are many codes in use such as repetition codes (such as the triple replication code considered earlier), parity-check codes where a parity symbol is attached, Hamming codes such as the (7, 4) code which has been applied for error correction of faulty memory.

   The Reed–Muller codes form a class of error-correcting codes that can correct more than one error. Cyclic codes are special linear block codes with efficient algebraic decoding algorithms. The BCH codes are an important class of cyclic codes, and the Reed–Solomon codes are an example of a BCH code.

   Convolution codes have been applied in the telecommunications field, for example in GSM, UMTS and in satellite communications. They belong to the class of linear codes, but also employ a memory so that the output depends on the current input symbols and previous input.

9.7   Review Questions

1. Describe the basic structure of a digital communication system.
2. Describe the mathematical structure known as the group. Give examples of groups, and explain the terms "commutative group" and "cyclic group". What is a normal group?
3. Describe the mathematical structure known as the ring and give examples of rings. Give examples of zero divisors in rings.
4. Describe the mathematical structure known as the field and give examples.
5. Describe the mathematical structure known as the vector space and give examples. Explain the terms linear independence and linear dependence.
6. Describe the encoding and decoding of an (n, k) block code where an information word of length k is converted to a codeword of length n.
7. Show how the minimum Hamming distance may be employed for error detection and error correction.
8. Describe linear block codes and show how a generator matrix may be employed to generate the codewords from the information words.

9.8   Summary

Coding theory is the branch of mathematics concerned with the reliable transmission of information over communication channels. It allows errors to be detected and corrected, and this is useful when messages are transmitted through a noisy communication channel. This branch of mathematics includes theory and practical algorithms for error detection and correction.

The theoretical foundations of coding theory are in abstract algebra including group theory, ring theory, fields and vector spaces.

An error-correcting code encodes the data by adding a certain amount of redundancy to the message so that the original message can be recovered if a small number of errors have occurred.

The fundamentals of block codes were discussed where an information word is of length k and a codeword is of length n. This led to the linear block codes $\mathbf{B}(n, k, d)$ and a discussion on error-detection and error-correction capabilities of the codes.

The goal of this chapter was to give a flavour of coding theory and the reader is referred to more specialised texts (e.g. [NFK:07]) for more detailed information.

Chapter 10
Language Theory and Semantics

> **Key Topics**
> Alphabets
> Grammars and Parse Trees
> Axiomatic Semantics
> Operational Semantics
> Denotational Semantics
> Lambda Calculus
> Lattices and Partial Orders

10.1 Introduction

There are two key parts to any programming language, and these are its syntax and semantics. The syntax is the grammar of the language and a program needs to be syntactically correct with respect to its grammar. The semantics of the language is deeper, and determines the meaning of what has been written by the programmer.

The difference between syntax and semantics may be illustrated by an example in a natural language. A sentence may be syntactically correct but semantically meaningless, or it may have semantic meaning but be syntactically incorrect. For example, consider the sentence:

> I will go to Dublin yesterday

This sentence is syntactically valid but semantically meaningless. Similarly, if a speaker utters the sentence "Me Dublin yesterday" we would deduce that the speaker had visited Dublin the previous day even though the sentence is syntactically incorrect.

The semantics of a programming language determines what a syntactically valid program will compute. A programming language is therefore given by:

$$\text{Programming Language} = \text{Syntax} + \text{Semantics}$$

Many programming languages have been developed over the last 60 years including *Plankalkul* which was developed by Zuse in the 1940s, *Fortran* by John Backus and others at IBM in the 1950s, *Cobol* was developed Grace Murray Hopper and the CODASYL committee in the late 1950s, *Algol 60* and *Algol 68* by an international commitee, *Pascal* by Wirth in the early 1970s, *Ada* was developed for the US military in the late 1970s, the C language developed by Richie and Thompson at Bell Labs in the 1970s, C++ by Stroustrup at Bell Labs in the early 1980s, and Java developed by Gosling at Sun Microsystems in the mid-1990s. There is a short account of a selection of programming languages in [ORg:12].

A programming language needs to have a well-defined syntax and semantics, and a compiler preserves the semantics of the language. Compilers are programs that translate a program written in some programming language into another form. It involves syntax analysis and parsing to check the syntactic validity of the program, semantic analysis to determine what the program should do, optimization to improve the speed and performance, and code generation in some target language.

Alphabets are a fundamental building block in language theory, as words and language are generated from alphabets. They are discussed in the next section.

10.2   Alphabets and Words

An *alphabet* is a finite non-empty set A, and the elements of A are called letters. For example, consider the set A which consists of the letters *a* to *z*.

Words are finite strings of letters, and a set of words is generated from the alphabet. For example, the alphabet $A = \{a, b\}$ generates the following set of words:

$$\{\varepsilon, a, b, aa, ab, bb, ba, aaa, bbb, \ldots \ldots \}$$

Each word consists of an ordered list of one or more letters and the set of words of length two consists of all ordered lists of two letters. It is given by

$$A^2 = \{aa, ab, bb, ba\}$$

Similarly, the set of words of length three is given by:

$$A^3 = \{aaa, aab, ab, aba, baa, bab, bbb, bba\}$$

The set of all words over the alphabet A is given by the positive closure A^+, and it is defined by:

$$A^+ = A \cup A^2 \cup A^3 \cup \cdots \cdots = \bigcup_{n=1}^{\alpha} A^n$$

Given any two words $w_1 = a_1 a_2 \ldots a_k$ and $w_2 = b_1 b_2 \ldots b_r$ then the concatenation of w_1 and w_2 is given by:

$$w = w_1 w_2 = a_1 a_2 \cdots a_k b_1 b_2 \cdots b_r$$

The empty word is a word of length zero and is denoted by ε. Clearly, $\varepsilon w = w\varepsilon = w$ for all w and so ε is the identity element under the concatenation operation. A^0 is used to denote the set containing the empty word $\{\varepsilon\}$, and the closure $A^* = (A^+ \cup \{\varepsilon\})$ denotes the set of all words over A (including empty words). It is defined as follows:

$$A^* = \overset{\alpha}{\underset{n=0}{\cup}} A^n$$

The mathematical structure $(A^*, {}^\wedge, \varepsilon)$ forms a monoid[1] where $^\wedge$ is the concatentation operator for words and the identity element is ε. The length of a word w is denoted by $|w|$ and the length of the empty word is zero, i.e. $|\varepsilon| = 0$.

A subset L of A^* is termed a formal language over A. Given two languages L_1, L_2 then the concatenation (or product) of L_1 and L_2 is defined by:

$$L_1 L_2 = \{w | w = w_1 w_2 \text{ where } w_1 \in L_1 \text{ and } w_2 \in L_2\}$$

The positive closure of L and the closure of L may also be defined as follows:

$$L^+ = \overset{\alpha}{\underset{n=1}{\cup}} L^n \quad L^* = \overset{\alpha}{\underset{n=0}{\cup}} L^n$$

A subset L of A^* is termed a formal language over A.

10.3   Grammars

A formal grammar describes the syntax of a language, and we distinguish between *concrete* and *abstract syntax*. Concrete syntax describes the external appearance of programs as seen by the programmer, whereas abstract syntax aims to describe the essential structure of programs rather than the external form. In other words, abstract syntax aims to give the components of each language structure while leaving out the representation details (e.g. syntactic sugar). Backus Naur Form (BNF) notation is often used to specify the concrete syntax of a language. A grammar consists of

- A finite set of terminal symbols
- A finite set of nonterminal symbols
- A set of production rules
- A start symbol

A formal grammar generates a formal language, which is a set of finite length sequences of symbols created by applying the production rules of the grammar. The application of a production rule involves replacing symbols at the left-hand side of the rule with the symbols on the right-hand side of the rule. The formal language then consists of all words consisting of terminal symbols that are reached by a derivation (i.e. the application of production rules) starting from the start symbol of the grammar.

---

[1] Recall that a monoid (M,*,e) is a structure that is closed and associative under the binary operation '*' and has identity element 'e'.
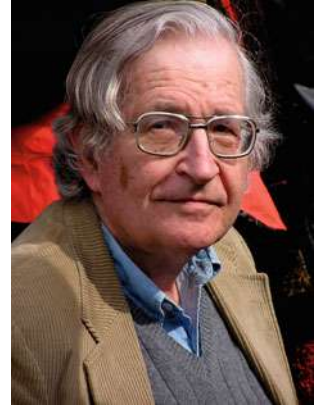
A construct that appears on the left-hand side of a production rule is termed a *non-terminal*, whereas a construct that only appears on the right-hand side of a production rule is termed a *terminal*. The set of non-terminals N is disjoint from the set of terminals A.

The theory of the syntax of programming languages is well established, and programming languages have a well-defined grammar that allows syntactically valid programs to be derived from the grammars.

Chomsky[2] classified a number of different types of grammar that occur (Fig. 10.1).

The Chomsky hierarchy consists of four levels including regular grammars, context free grammars, context sensitive grammars and unrestricted grammars. The grammars are distinguished by the production rules, which determine the type of language that is generated (Table 10.1).

Regular grammars are used to generate the words that may appear in a programming language. These includes the identifiers (e.g. names for variables, functions and procedures), special symbols (e.g. addition, multiplication, etc.) and the reserved words of the language.

A rewriting system for context free grammars is a finite relation between N and $(A \cup N)^*$, i.e. a subset of $N \times (A \cup N)^*$, a production rule $<N> \to w$ is one element of this relation, and is an ordered pair $(<N>, w)$ where w is a word consisting of zero or more terminal and non-terminal letters. This production rule means that $<N>$ may be replaced by w.

10.3.1 Backus Naur Form

Backus–Naur Form[3] (BNF) provides an elegant means of specifying the syntax of programming languages. It was originally employed to define the grammar for the

---

[2] Chomsky made important contributions to linguistics and the theory of grammars. He is more widely known to-day as a critic of US foreign policy.

[3] Backus–Naur Form is named after John Backus and Peter Naur. It was created as part of the design of the Algol 60 programming language, and is used to define the syntax rules of the language.

Table 10.1 Chomsky hierarchy of grammars

Grammar type	Description
Type 0 Grammar	Type 0-grammars include all formal grammars. They have production rules of the form $\alpha \to \beta$ where α and β are strings of terminals and non-terminals. They generates all languages that can be recognized by a Turing machine
Type 1 Grammar (context sensitive)	These grammars generate the context sensitive languages. They have production rules of the form $\alpha A\beta \to \alpha\gamma\beta$ where A is a non-terminal and α, β and γ are strings of terminals and non-terminals. A linear bounded automaton recognizes these languages[a]
Type 2 Grammar (context free)	These grammars generate the context free languages. These are defined by rules of the form $A \to \gamma$ where A is a non-terminal and γ is a string of terminals and non-terminals. These languages are recognized by a pushdow automaton[b] and are used to define the syntax of most programming languages
Type 3 Grammar (regular grammars)	These grammars generate the regular languages (or regular expressions). These are defined by rules of the form $A \to a$ or $A \to aB$ where A and B are non-terminals and a is a single terminal. A finite state automaton recognizes these languages, and regular expressions are used to define the lexical structure of programming languages

[a]A linear bounded automaton is a restricted form of a nondeterministic Turing machine in which a limited finite portion of the tape (a function of the length of the input) may be accessed.
[b]A pushdown automaton is a finite automaton that can make use of a stack containing data.

Algol-60 programming language, and a variant was used by Wirth to specify the syntax of the Pascal programming language. BNF is widely used today to specify the syntax of programming languages.

A BNF specification essentially describes the external appearance of a program as seen by the programmer. The grammar of a context-free grammar may then be input into a parser (e.g. Yacc), and the parser is used to determine if a program is syntactically correct or not.

A BNF specification consists of a set of production rules with each production rule describing the form of a class of language elements such as expressions, statements, and so on. A production rule is of the form:

$$<\text{symbol}> :: = <\text{expression with symbols}>$$

where <symbol> is a *nonterminal*, and the expression consists of a sequence of terminal and nonterminal symbols. A construct that has alternate forms appears more than once, and this is expressed by sequences separated by the vertical bar 'l' (which indicates a choice). In other words, there is more than one possible substitution for the symbol on the left-hand side of the rule. Symbols that never appear on the left-hand side of a production rule are called *terminals*.

The following is the partial BNF definition of the syntax of various statements in a ssample programming language:

$$<\text{loop statement}> :: = <\text{while loop}>|<\text{for loop}>$$

<while loop> :: = while (<condition>)<statement>

<for loop> :: = for (<expression>)<statement>

<statement> :: = <assignment statement>|<loop statement>

<assignment statement> :: = <variable> : = <expression>

It includes various non-terminals such as (<loop statement>, <while loop>, and so on. The terminals include 'while', 'for', ': = ', '(' and ')'. The production rules for <condition> and <expression> are not included.

The grammar of a context-free language (e.g. LL(1), LL(k), LR(1), LR(k) grammar expressed in BNF notation) may be translated by a parser into a parse table. The parse table may then be employed to determine whether a particular program is valid with respect to its grammar.

Example 10.1 (Context-free grammar) This example considers a context free grammar for parenthesis matching in which there are two terminal symbols and one non-terminal symbol.

$$S \rightarrow SS$$
$$S \rightarrow (S)$$
$$S \rightarrow ()$$

Then by starting with S and applying the rules we can construct:

$$S \rightarrow SS \rightarrow (S)S \rightarrow (())S \rightarrow (())()$$

Example 10.2 (Context-free grammar) This example considers the definition of a context-free grammar for expressions in a programming langage. The definition is ambiguous as there is more than one derivation tree for some expressions (e.g. there are two parse trees for the expression $5 \times 3 + 1$ as discussed below).
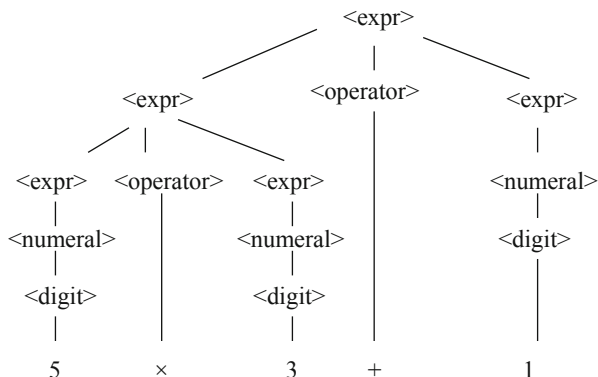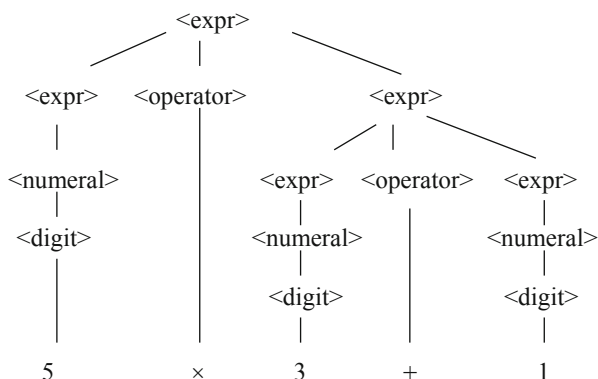
<expr> :: = <numeral> | (<expr>)
                    | (<expr> <operator> <expr>)
<operator> :: = + | − | × | /
    <digit> :: = 0 | 1 | ... | 9
<numeral> :: = <digit> | <digit> <numeral>

Example 10.3 (Regular grammar) The definition of an identifier in most programming languages is similar to:

<identifier> :: = <let> <letdig>
    <letdig> :: = <let> | <dig> | ε
    <letdig> :: = <let> <letdig> | <dig> <letdig>
        <let> :: = a | b | c | ... | z
        <dig> :: = 0 | 1 | ... | 9

10.3.2   Parse Trees and Derivations

Let A and N be the terminal and nonterminal alphabet of a rewriting system and let <X> $\rightarrow w$ be a production. Let x be a word in $(A \cup N)$* with $x = u$ <X> v for some

Fig. 10.2 Parse tree $5 \times 3 + 1$

Fig. 10.3 Parse Tree
$5 \times 3 + 1$

words $u, v \in (A \cup N)^*$. Then x is said to directly yield uwv and this is written as $x \Rightarrow uwv$.

This single substitution ($\Rightarrow$) can be extended by a finite number of productions ($\Rightarrow^*$), and this gives the set of words that can be obtained from a given word. This derivation is achieved by applying several production rules (one production rule is applied at a time) in the grammar.

That is, given $x, y \in (A \cup N)^*$, x yields y (or y is a derivation of x) if $x = y$, or there exists a sequence of words $w_1, w_2, \ldots, w_n \in (A \cup N)^*$ such that $x = w_1, y = w_n$ and $w_i \Rightarrow w_{i+1}$ for $1 \leq i \leq n-1$. This is written as $x \Rightarrow^* y$.

The expression grammar presented in Example 10.2 is ambiguous, and this means that an expression such as $5 \times 3 + 1$ has more than one interpretation. It is not clear from the grammar whether multiplication is performed first and then addition, or whether addition is performed first and then multiplication.

There are two parse trees for the expression $5 \times 3 + 1$ (Fig. 10.2 and Fig. 10.3). The interpretation of the first parse tree is that multiplication is performed first and then addition (this is the normal interpretation of such expressions in programming languages as multiplication is a higher precedence operator than addition).

The interpretation of the second parse tree is that addition is performed first and then multiplication (Fig. 10.3). It may seem a little strange that one expression has two parse trees and it shows that the grammar is ambiguous. This means that there is a choice for the compiler in evaluating the expression and needs to assign the right meaning to the expression. One solution would be for the language designer to alter the definition of the grammar to remove the ambiguity.

10.4   Programming Language Semantics

The formal semantics of a programming language is concerned with defining the actual meaning of a language. Language semantics is deeper than syntax, and the theory of the syntax of programming languages is well established. A programmer writes a program according to the rules of the language. The compiler first checks the program for synyactic correctness: i.e., it determines whether the program as written is valid according to the rules of the grammar of the language. If the program is syntactically correct, then the compiler determines the meaning of what has been written and generates the corresponding machine code.[4]

The compiler must preserve the semantics of the language, i.e. the semantics are not defined by the compiler, but rather the function of the compiler is to preserve the semantics of the language. Therefore, there is a need to have an unambiguous definition of the meaning of the language independently of the compiler, and the meaning is then preserved by him.

A program's syntax[5] gives no information as to the meaning of the program, and therefore there is a need to supplement the syntactic description of the language with a formal unambiguous definition of its semantics.

It is possible to utter syntactically correct but semantically meaningless sentences in a natural language. Similarly, it is also possible to write syntactically correct programs that behave in quite a different way from the intention of the programmer.

The formal semantics of a language is given by a mathematical model that describes the possible computations described by the language. There are three main approaches namely axiomatic semantics, operational semantics and denotational semantics (Table 10.2).

There are several applications of programming language semantics including language design, program verification, compiler writing and language standardisation. The three main approaches to semantics are described in more detail below.

---

[4] Of course, what the programmer has written may not be what the programmer had intended.

[5] There are attribute (or affix) grammars that extend the syntactic description of the language with supplementary elements covering the semantics. The process of adding semantics to the syntactic description is termed decoration.

Table 10.2 Programming language semantics

Approach	Description
Axiomatic semantics	This involves giving meaning to phrases of the language using logical axioms
	It employs *pre-* and *post- condition assertions* to specify what happens when the statement executes. The relationship between the initial assertion and the final assertion essentially gives the semantics of the code
Operational semantics	This approach describes how a valid program is interpreted as *sequences of computational steps*. These sequences then define the meaning of the program
	An abstract machine (SECD machine) may be defined to give meaning to phrases, and this is done by describing the transitions they induce on states of the machine
Denotational semantics	This involves defining the meaning of programs in terms of mathematical objects such as integers, tuples and functions
	Each phrase in the language is translated into a mathematical object that is termed the *denotation* of the phrase

10.4.1   Axiomatic Semantics

Axiomatic semantics gives meaning to phrases of the language by describing the logical axioms that that apply to them. It was developed by C.A.R. Hoare[6] in a famous paper '*An axiomatic basis for computer programming*' [Hor:69]. His axiomatic theory consists of *syntactic elements*, *axioms* and *rules of inference*.

The well-formed formulae that are of interest in axiomatic semantics are pre- and post assertion formulae of the form $P\{a\}Q$, where a is an instruction in the language and P and Q are assertions: i.e., properties of the program objects that may be true or false.

An *assertion* is essentially a predicate that may be true in some states and false in other states. For example, the assertion $(x - y > 5)$ is true in the state in which the values of x and y are 7 and 1 respectively, and false in the state where x and y have values 4 and 2.

The pre- and post-condition assertions are employed to specify what happens when the statement executes. The relationship between the initial assertion and the final assertion gives the semantics of the code statement. The *pre-* and *post- condition* assertions are of the form:

$$p\{a\}Q$$

The pre-condition P is a predicate (input assertion), and the post-condition Q is a predicate (output assertion). The braces separate the assertions from the program fragment. The well-formed formula $P\{a\}Q$ is itself a predicate that is either true or false.

---

[6] Hoare was influenced by earlier work by Floyd on assigning meanings to programs using flowcharts [Flo:67].

This notation expresses the *partial correctness*[7] of *a* with respect to *P* and *Q*, and its meaning is that if statement *a* is executed in a state in which the predicate *P* is true and execution terminates, then it will result in a state in which assertion *Q* is satisfied.

The axiomatic semantics approach is described in more detail in [ORg:06], and the axiomatic semantics of a selection of statements is presented below.

- *Skip*: The skip statement does nothing and whatever condition is true on entry to the command is true on exit from the command. Its meaning is given by:

$$P\{skip\}P$$

- *Assignment*: The meaning of the assignment statement is given by the axiom:

$$P^x_e\{x := e\}P$$

The meaning of the assignment statement is that *P* will be true after execution of the assignment statement if and only if the predicate P^x_e with the value of *x* replaced by *e* in *P* is true before execution (since *x* will contain the value of *e* after execution).

The notation P^x_e denotes the expression obtained by substituting *e* for all free occurrences of *x* in *P*.

- *Compound*: The meaning of the conditional command is:

$$\frac{P\,\{S_1\}\,Q,\ Q\,\{S_2\}\,R}{P\,\{S_1;\ S_2\}\,R}$$

The compound statement involves the execution of S_1 followed by the execution of S_2. The meaning of the compound statement is that *R* will be true after the execution of the compound statement S_1; S_2 provided *P* is true, if it is established that *Q* will be true after the execution of S_1 provided that *P* is true, and that *R* is true after the execution of S_2 provided *Q* is true.

There needs to be at least one rule associated with every construct in the language in order to give its axiomatic semantics. The semantics of other programming language statements such as the 'while' statement and the 'if' statement are described in [ORg:06].

10.4.2 Operational Semantics

The operational semantics definition is similar to an interpreter, where the semantics of a language are expressed by a mechanism that makes it possible to determine the effect of any program in the language. The meaning of a program is given by

---

[7] Total correctness is expressed using {P}a{Q} and program fragment a is totally correct for precondition P and postcondition Q if and only if whenever a is executed in any state in which P is satisfied then execution terminates, and the resulting state satisfies Q.

the evaluation history that an interpeter produces when it interprets the program. The interpreter may be close to an executable programming language or it may be a mathematical language.

The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. The evaluation history then defines the meaning of the program, and is a sequence of internal interpreter configurations.

One early use of operational semantics was the work done by John McCarthy in the late 1950s on the semantics of LISP in terms of the lambda calculus. The use of lambda calculus allows the meaning of a program to be expressed using a mathematical interpreter, and this offers precision through the use of mathematics.

The meaning of a program may be given in terms of a hypothetical or virtual machine that performs the set of actions that corresponds to the program. An abstract machine (SECD machine)[8] may be defined to give meaning to phrases in the language, and this is done by describing the transitions that they induce on states of the machine.

Operational semantics give an intuitive description of the programming language being studied, and its descriptions are close to real programs. It can play a useful role as a testing tool during the design of new languages, as it is relatively easy to design an interpreter to execute the description of example programs. This allows the effects of new languages or new language features to be simulated and studied through actual execution of the semantic descriptions prior to writing a compiler for the language. In other words, operational semantics can play a role in rapid prototyping during language design, and to get early feedback on the suitability of the language.

One disadvantage of the operational approach is that the meaning of the language is understood in terms of execution, i.e. in terms of interpreter configurations, rather than in an explicit *machine independent specification*. An operational description is just one way to execute programs. Another disadvantage is that the interpreters for non-trivial languages often tend to be large and complex.

A more detailed account of operational semantics is in [Plo:81, Mey:90].

10.4.3   Denotational Semantics

Denotational semantics [Mey:90] expresses the semantics of a programming language by a translation schema that associates a meaning (*denotation*) with each program in the language. It maps a program directly to its meaning, and it was originally called mathematical semantics as it provides meaning to programs in terms of mathematical values such as integers, tuples and functions. That is, the meaning of a program is a mathematical object, and an interpreter is not employed. Instead, a valuation function is employed to map a program directly to its meaning, and the

---

[8] This virtual stack based machine was originally designed by Peter Landin to evaluate lambda calculus expres-sions, and it has since been used as a targefor several compilers.
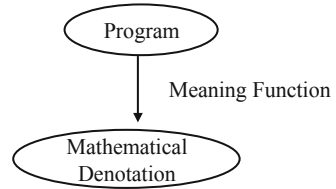
denotational description of a programming language is given by a set of *meaning functions M* associated with the constructs of the language (Fig. 10.4).

Each meaning function is of the form $M_T: T \to D_T$ where T is some construct in the language. Many of the meaning functions will be 'higher-order', i.e. functions that yield functions as results. The signature of the meaning function is from syntactic domains (i.e. T) to semantic domains (i.e. D_T). A valuation map $V_T: T \to B$ may be employed to check the static semantics prior to giving a meaning of the language construct.[9]

A denotational definition is more abstract than an operational definition. It does not specify the computational steps, and its exclusive focus is on the programs to the exclusion of the state and other data elements. The state is less visible in denotational specifications.

It was developed by Christopher Strachey and Dana Scott at the Programming Research Group at Oxford, England in the mid-1960s, and their approach to semantics is known as the Scott–Strachey approach [Sto:77]. It provided a mathematical foundation for the semantics of programming languages.

Dana Scott's contributions included the formulation of domain theory, and this allowed programs containing recursive functions and loops to be given a precise semantics. Each phrase in the language is translated into a mathematical object that is the *denotation* of the phrase. Denotational semantics has been applied to language design and implementation.

10.5   Lambda Calculus

Functions are an essential part of mathematics and play a key role in specifying the semantics of programming language constructs. Functions are a special type of relations, and they were discussed in Chap. 2. Simple finite functions may be defined as an explicit set of pairs, e.g.

$$f \triangleq \{(a, 1), (b, 2), (c, 3)\}$$

However, for more complex functions there is a need to define the function more abstractly, rather than listing all of its member pairs. This may be done in a manner

---

[9] This is similar to what a compiler does in that if errors are found during the compilation phase, the compiler halts and displays the errors and does not continue with code generation.

similar to set comprehension, where a set is defined in terms of a characteristic property of its members.

Functions may be defined by comprehension through a powerful abstract notation known as lambda calculus. This notation was introduced by Alonzo Church in the 1930s to study computability, and lambda calculus provides an abstract framework for describing mathematical functions and their evaluation. It may be used to study function definition, function application, parameter passing and recursion.

Any computable function can be expressed and evaluated using lambda calculus or Turing machines, as these are equivalent formalisms. Lambda calculus uses a small set of transformation rules, and these include:

- Alpha-conversion rule (α-conversion)[10]
- Beta-reduction rule (β-reduction)[11]
- Eta-conversion (η-conversion)[12]

Every expression in the λ**-**calculus stands for a function with a single argument. The argument of the function is itself a function with a single argument and so on. The definition of a function is anonymous in the calculus. For example, the function that adds one to its argument is usually defined as $f(x) = x + 1$. However, in λ**-**calculus the function is defined as:

$$succ \underline{\Delta} \lambda \ x.x + 1$$

The name of the formal argument x is irrelevant and an equivalent definition of the function is $\lambda \ z. \ z + 1$. The evaluation of a function f with respect to an argument (e.g. 3) is usually expressed by $f(3)$. In λ**-**calculus this would be written as $(\lambda \ x. \ x + 1) \ 3$, and this evaluates to $3 + 1 = 4$. Function application is left associative, i.e. $f \ x \ y = (f \ x) \ y$. A function of two variables is expressed in lambda calculus as a function of one argument, which returns a function of one argument. This is known as *currying* (named after the American logician, Haskell B. Curry), e.g. the function $f(x, y) = x + y$ is written as $\lambda \ x. \ \lambda \ y. \ x + y$. This is often abbreviated to $\lambda \ x \ y. \ x + y$.

Lambda-calculus is a simple mathematical system, and its syntax is defined as follows:

$$<exp> :: = <identifier> |$$
$$\lambda <identifier>.<exp> | \text{ -- abstraction}$$
$$<exp><exp> | \text{ -- application}$$
$$(<exp>)$$

The four lines of syntax plus *conversion* rules of Lambda-calculus's are sufficient to define Booleans, integers, data structures and computations on them. It inspired Lisp

---

[10] This essentially expresses that the names of bound variables is unimportant.

[11] This essentially expresses the idea of function application.

[12] This essentially expresses the idea that two functions are equal if and only if they give the same results for all arguments.

and modern functional programming languages. The original calculus was untyped, but typed lambda calculi have been introduced in recent years. The typed lambda calculus allows the sets to which the function arguments apply to be specified. For example, the definition of the *plus* function is given as:

$$plus \triangleq \lambda a, b : \mathbb{N} \cdot a + b$$

The lambda calculus makes it possible to express properties of the function without reference to members of the base sets on which the function operates. It allows operations such as function composition to be applied, and it also provides powerful support for higher order functions. This is important in the expression of the denotational semantics of the constructs of programming languages.

10.6   Lattices and Order

This section considers some of the mathematical structures used in the definition of the semantic domains used in denotational semantics. These mathematical structures may be employed to give a secure foundation for recursion, and it is natural to ask, when presented with a recursive definition, whether it means anything at all, and in some cases the answer is negative. It is therefore important to understand when recursion may be used safely.

Recursive definitions are a powerful and elegant way of giving the denotational semantics of language constructs. The mathematical structures considered in this section include partial orders, total orders, lattices, complete lattices and complete partial orders.

10.6.1   Partially Ordered Sets

A *partial order* $\leq$ on a set P is a binary relation such that for all $x, y, z \in P$ the following properties hold:

  (i) $x \leq x$ (reflexivity)
 (ii) $x \leq y$ and $y \leq x \Rightarrow x = y$ (anti-symmetry)
(iii) $x \leq y$ and $y \leq z \Rightarrow x \leq z$ (transitivity)

A set P with an order relation $\leq$ is said to be a *partially ordered* set.

Example 10.4 Consider the powerset $\mathbb{P}X$ that consists of all the subsets of the set X with the ordering defined by set inclusion. That is, $A \leq B$ if and only if $A \subseteq B$ then $\subseteq$ is a partial order on $\mathbb{P}X$.

A partially ordered set is a *totally ordered* set (*also called chain*) if for all $x, y \in P$ then either $x \leq y$ or $y \leq x$. That is, any two elements of P are directly comparable.

A partially ordered set P is an *anti-chain* if for any x, y in P then $x \leq y$ only if $x = y$. That is, the only elements in P that are comparable to a particular element are the element itself.

Maps between Ordered Sets Let P and Q be partially ordered sets then a map ϕ from P to Q may preserve the order in P and Q. We distinguish between order preserving, order embedding and order isomorphism. These terms are defined as follows:

Order preserving (or *monotonic increasing function*)
A mapping $\phi: P \to Q$ is said to be order preserving if

$$x \leq y \Rightarrow \phi(x) \leq \phi(y)$$

Order embedding
A mapping $\phi: P \to Q$ is said to be an order embedding if

$$x \leq y \text{ in } P \text{ if and only if } \phi(x) \leq \phi(y) \text{ in } Q.$$

Order isomorphism
The mapping $\phi: P \to Q$ is an order isomorphism if and only if it is an order embedding mapping onto Q.

Dual of a Partially Ordered Set The dual of a partially ordered set P (denoted P^{∂}) is a new partially ordered set formed from P where $x \leq y$ holds in P^{∂} if and only if $y \leq x$ holds in P (i.e. P^{∂} is obtained by reversing the order on P).

For each statement about P there corresponds a statement about P^{∂}. Given any statement Φ about a partially ordered set, then the dual statement Φ^{∂} is obtained by replacing each occurrence of $\leq$ by $\geq$ and vice versa.

Duality Principle Given that statement Φ is true of a partially ordered set P, then the statement Φ^{∂} is true of P^{∂}.

Maximal and Minimum Elements Let P be a partially ordered set and let $Q \subseteq P$ then

 (i) $a \in Q$ is a *maximal* element of Q if $a \leq x \in Q \Rightarrow a = x$.
(ii) $a \in Q$ is the *greatest (*or *maximum) element of Q if $a \geq x$ for every $x \in Q$ and in that case we write $a = \max Q$

A *minimal* element of Q and the *least (*or *minimum)* are defined dually by reversing the order. The greatest element (if it exists) is called the top element and is denoted by $\top$. The least element (if it exists) is called the bottom element and is denoted by $\bot$.

Example 10.5 Let X be a set and consider $\mathbb{P}X$ the set of all subsets of X with the ordering defined by set inclusion. The top element $\top$ is given by X, and the bottom element $\bot$ is given by $\emptyset$.

A finite totally ordered set always has top and bottom elements, but an infinite chain need not have.

10.6.2 *Lattices*

Let P be a partially ordered set and let $S \subseteq P$. An element $x \in P$ is an upper bound of S if $s \leq x$ for all $s \in S$. A lower bound is defined similarly.

The set of all upper bounds for S is denoted by S^u, and the set of all lower bounds for S is denoted by S^l.

$$S^u = \{x \in P|(\forall s \in S) s \geq x\}$$
$$S^l = \{x \in P|(\forall s \in S s \geq x\})$$

If S^u has a least element x then x is called the *least upper bound* of S. Similarly, if S^l has a greatest element x then x is called the *greatest lower bound* of S.

In other words, x is the least upper bound of S if

 (i) x is an upper bound of S.
(ii) $x \leq y$ for all upper bounds y of S

The least upper bound of S is also called the *supremum* of S denoted (sup S), and the greatest lower bound is also called the infimum of S, and is denoted by inf S.

Join and Meet Operations The join of x and y (denoted by $x \vee y$) is given by sup $\{x, y\}$ when it exists. The meet of x and y (denoted by $x \wedge y$) is given by inf$\{x, y\}$ when it exists.

The supremum of S is denoted by $\vee S$, and the infimum of S is denoted by $\wedge S$.

Definition Let P be a non-empty partially ordered set then

1. If $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$ then P is called a *lattice*.
2. If $\vee S$ and $\wedge S$ exist for all $S \subseteq P$ then P is called a *complete lattice*

Any complete lattice is bounded (i.e. it has top and bottom elements)

Example 10.6 Let X be a set and consider $\mathbb{P}X$ the set of all subsets of X with the ordering defined by set inclusion. Then $\mathbb{P}X$ is a complete lattice in which

$$\vee \{A_i|i \in I\} = \cup A_i$$
$$\wedge \{A_i|i \in I\} = \cap A_i$$

10.6.3 *Complete Partial Orders*

Let S be a non-empty subset of a partially ordered set P. Then

1. S is said to be a *directed set* if for every finite subset F of S there exists $z \in S$ such that $z \in F^u$.
2. S is said to be *consistent* if for every finite subset F of S there exists $z \in P$ such that $z \in F^u$

A partially ordered set P is a *complete partial order* (CPO) if:

1. *P* has a bottom element ⊥
2. ∨*D* exists for each directed subset *D* of *P*

The simplest example of a directed set is a chain, and we note that any complete lattice is a complete partial order, and that any finite lattice is a complete lattice.

10.6.4   Recursion

Recursive definitions arise frequently in programs and offer an elegant way to define routines and data types. A recursive routine contains a direct or indirect call to itself, and a recursive data type contains a direct or indirect reference to specimens of the same type. Recursion needs to be used with care, as there is always a danger that the recursive definition may be circular (i.e. defines nothing). It is therefore important to investigate when a recursive defintion may be used safely, and to give a mathematical definition of recursion.

The control flow in a recursive routine must contain at least one non-recursive branch, since if all possible branches include a recursive form the routine could never terminate. Further, the value of at least one argument in the recursive call must be different from the initial value of the formal argument, as otherwise the recursive call would result in the same sequence of events, and therefore would never terminate.

The mathematical meaning of recursion is defined in terms of *fixed point theory*, which is concerned with determining solutions to equations of the form $x = \tau(x)$, where the function τ is of the form $\tau: X \to X$.

A recursive definition may be interpreted as a fixpoint equation of the form $f = \Phi(f)$; i.e. the fixpoint of a high-level functional Φ that takes a function as an argument. For example, consider the functional Φ defined as follows:

$$\Phi \underline{\Delta} \lambda f \ \lambda n \cdot \text{if } n = 0 \text{ then } 1 \text{ else } n^* f \ (n-1)$$

Then a fixpoint of Φ is a function f such that $f = \Phi(f)$ or in other words

$$f = \lambda n \cdot \text{if } n = 0 \text{ then } 1 \text{ else } n^* f (n-1)$$

Clearly, the factorial function is a fixpoint of Φ, and it is the only total function that is a fixpoint. The solution of the equation $f = \Phi(f)$ (where Φ has a fixpoint) is determined as the limit f of the sequence of functions $f_0, f_1, f_2, \ldots$, where the f_i are defined inductively as:

$$f_0 \ \underline{\Delta} \ \varnothing \qquad \text{(the empty partial function)}$$

$$f_i \underline{\Delta} \Phi \left(f_{i-1} \right)$$

Each f_i may be viewed as a successive approximation to the true solution f of the fixpoint equation, with each f_i bringing a little more information on the solution than its precessor f_{i-1}.

The function f_i is defined for one more value than f_{i-1}, and gives the same result for any value for which they are both defined. The definition of the factorial function is thus built up as follows:

$$f_0 \underline{\Delta} \varnothing \qquad\qquad\qquad \text{(the empty partial function)}$$

$$f_1 \underline{\Delta} \{0 \to 1\}$$

$$f_2 \underline{\Delta} \{0 \to 1, 1 \to 1\}$$

$$f_3 \underline{\Delta} \{0 \to 1, 1 \to 1, 2 \to 2\}$$

$$f_4 \underline{\Delta} \{0 \to 1, 1 \to 1, 2 \to 2, 3 \to 6\}$$

$$\vdots \qquad\qquad\qquad \vdots$$

$$\vdots$$

For every i, the domain of f_i is the interval 1, 2, ..., $i-1$ and $f_i(n) = n!$ for any n in this interval. In other words f_i is the factorial function restricted to the interval 1, 2, ..., $i-1$. The sequence of f_i may be viewed as successive approximations of the true solution of the fixpoint equation (which is the factorial function), with each f_i bringing defined for one more value that its predessor f_{i-1}, and defining the same result for any value for which they are both defined.

The candidate fixpoint f_∞ is the limit of the sequence of functions f_i, and is the union of all the elements in the sequence. It may be written as follows:

$$f_\infty \underline{\Delta} \varnothing \cup \Phi(\varnothing) \cup \Phi(\Phi(\varnothing)) \cup \cdots = \cup_{i:\mathbb{N}} f_i$$

where the sequence f_i is defined inductively as

$$f_0 \underline{\Delta} \varnothing \qquad\qquad \text{(the empty partial function)}$$

$$f_{i+1} \underline{\Delta} f_i \cup \Phi(f_i)$$

This forms a subset chain where each element is a subset of the next, and it follows by induction that:

$$f_{i+1} = \cup_{j:0,\,\ldots,\,i} \Phi(f_i)$$

A general technique for solving fixpoint equations of the form $h = \tau(h)$ for some functional τ is to start with the least defined function $\varnothing$ and iterate with τ. The union of all the functions obtained as successive sequence elements is the fixpoint.

The conditions in which f_∞ is a fixpoint of Φ is the requirement for $\Phi(f_\infty) = f_\infty$. This is equivalent to:

$$\Phi(\cup_{i:\mathbb{N}} f_i) = \cup_{i:\mathbb{N}} f_i$$

$$\Phi(\cup_{i:\mathbb{N}} f_i) = \cup_{i:\mathbb{N}} \Phi(f_i)$$

A sufficient point for Φ to have a fixpoint is that the property $\Phi(\cup_{i:\mathbb{N}} f_i) = \cup_{i:\mathbb{N}} \Phi(f_i)$ holds for any subset chain f_i.

A detailed account on the mathematics of recursion is given in Chap. 8 of [Mey:90].

10.7 Review Questions

1. Explain the difference between syntax and semantics.
2. Describe the Chomsky hierarchy of grammars and give examples of each type.
3. Show that a grammar may be ambiguous leading to two difference parse trees. What problems does this create and how should it be dealt with?
4. Describe axiomatic semantics, operation semantics and denotational semantics and expain the differences between them.
5. Explain partial orders, lattices and complete partial orders. Give examples of each.
6. Show how the meaning of recursion is defined with fixpoint theory.

10.8 Summary

This chapter considered two key parts to any programming language namely syntax and semantics. The syntax of the language is concerned with the production of grammatically correct programs in the language, whereas the semantics of the language is deeper and is concerned with the meaning of what has been written by the programmer.

There are several approaches to defining the semantics of programming languages, and these include axiomatic, operational and denotational semantics. Axiomatic semantics is concerned with defining properties of the language in terms of axioms; operational semantics is concerned with defining the meaning of the language in terms of an interpreter; and denotational semantics is concerned with defining the meaning of the phrases in a language in terms of mathematical functions.

Compilers are programs that translate a program written in some programming language into another form. It involves syntax analysis and parsing to check the syntactic validity of the program, semantic analysis to determine what the program should do, optimization to improve the speed and performance of the compiler, and code generation in some target language. Various mathematical structures including partial orders, total orders, lattices and complete partial orders were considered. These are useful building blocks in the definition of the denotational semantics of a language, and in giving a mathematical interpretation of recursion.

Chapter 11
Computability and Decidability

Key Topics
    Computability
    Completeness
    Decidability
    Formalism

11.1 Introduction

It is impossible for a human being or a machine to write out all of the members of an infinite countable set, such as the set of natural numbers $\mathbb{N}$. However, humans can do something quite useful in the case of certain enumerable infinite sets: They can give explicit instructions (that may be followed by a machine or another human being) to produce the nth member of the set for an arbitrary finite n. The problem remains that for all but a finite number of values of n it will be physically impossible for any human being or a machine to actually carry out the computation, due to the limitations on the time available for computation, the speed at which the individual steps in the computation may be carried out, and due to finite materials.

The intuitive meaning of computability is in terms of an algorithm (or effective procedure) that specifies a set of instructions to be followed to complete the task. In other words, a function f is *computable* if there exists an algorithm that produces the value of f correctly for each possible argument of f. The computation of f for a particular argument x just involves following the instructions in the algorithm, and it produces the result $f(x)$ in a finite number of steps if x is in the domain of f. If x is not in the domain of f, then the algorithm may produce an answer saying so, or it might run forever never halting. A computer program implements an algorithm.

The concept of computability may be made precise in several equivalent ways such as Church's *lambda calculus*, *recursive function theory* or by the theoretical

Turing machines.[1] These are all equivalent, and perhaps the most well known is the Turing machine. This is a mathematical machine with a potentially infinite tape divided into frames (or cells) in which very basic operations can be carried out. The set of functions that are computable are those that are computable by a Turing machine.

Decidability is an an important topic in modern mathematics. Church and Turing independently showed in 1936 that mathematics is not decidable. In other words, there is no mechanical procedure (i.e. algorithm) to determine whether an arbitrary mathematical proposition is true or false, and so the only way to determine the truth or falsity of a statement is to try to solve the problem. The fact that there is no general method to solve all instances of a specific problem, as well as the impossibility of proving or disproving certain statements within a formal system may suggest limitations of human and machine knowledge.

11.2 Formalism

Gottlob Frege (a nineteenth century German mathematician and logician) invented a formal system which is the basis of modern predicate logic. It included axioms, definitions, universal and existential quantification, and formalization of proof. His objective was to show that mathematics was reducible to logic but his project failed as one of the axioms that he had added to his system proved to be inconsistent. This inconsistency was pointed out by Bertrand Russell, and is known as *Russell's paradox.*[2]

The expressions in a formal system are terms, and a term may be simple or complex. A simple term may be an object such as a number, and a complex term may be an arithmetic expression such as $4^3 + 1$. A complex term is formed via functions, and the expression above uses two functions, namely the cube function with argument 4 and the plus function with two arguments.

The sentences of the logical system denote the truth-values of true or false. The sentences may include expressions such as equality ($x = y$), and this returns true if x is the same as y, and false otherwise. Similarly, a more complex expression such as $f(x, y, z) = w$ is true if $f(x, y, z)$ is identical with w, and false otherwise. Frege represented statements such as '5 is a prime' by '$P(5)$' where $P(x)$ is termed a concept. The statement $P(x)$ returns true if x is prime. His approach was to represent a predicate as a function of one variable which returns a Boolean value of true or false.

Formalism was proposed by Hilbert as a foundation for mathematics in the early twentieth century. The motivationwas to provide a secure foundation for mathematics, and to resolve the contradictions in the formalisation of set theory identified by Russell's paradox. The presence of a contradiction in a theory means the collapse

---

[1] The Church-Turing thesis states that anything that is computable is computable by a Turing machine.

[2] Russell's paradox considers the question as to whether the set of all sets that contain themselves as members is a set. In either case there is a contradiction.

Fig. 11.1 David Hilbert

of the whole theory, and so it was seen as essential that there be a proof of the consistency of the formal system. The methods of proof in mathematics are formalised with axioms and rules of inference.

A formal system contains meaningless symbols together with rules for manipulating them. The individual formulae are certain finite sequences of symbols obeying the syntactic rules of the formal language. A formal system consists of:

- A formal language
- A set of axioms
- Rules of inference.

A formal system is generally intended to represent some aspect of the real world. A *rule of inference* relates a set of formulae $(P_1, P_2, \ldots, P_k)$ called the premises to the consequence formula Q called the conclusion. For each rule of inference there is a finite procedure for determining whether a given formula Q is an immediate consequence of the rule from the given formulas $(P_1, P_2, \ldots, P_k)$. A *proof* in a formal system consists of a finite sequence of formulae, where each formula is either an axiom or derived from one or more preceding formulae in the sequence by one of the rules of inference (Fig. 11.1).

Hilbert's program was concerened with the formalisation of mathematics (i.e., the axiomatization of mathematics) together with a proof that the axiomatization was consistent. The specific objectives of Hilbert's program were to:

- Provide a secure foundation for mathematics by a formalisation of mathematics.
- Show that the formalisation of mathematics is *complete*: i.e. all mathematical truths can be proved in the formal system.
- Provide a proof that the formal system is *consistent* (i.e. no contradictions may be derived).
- Show that mathematics is *decidable* i.e. there is an algorithm to determine the truth of falsity of any mathematical statement.

The formalist movement in mathematics led to the formalisation of large parts of mathematics, where theorems could be proved using just a few mechanical rules. The

two most comprehensive formal systems developed were *Principia Mathematica* by Russell and Whitehead, and the axiomatisation of the set theory by Zermelo–Fraenkel (subsequently developed further by von Neumann).

Principia Mathematica is a comprehensive three volume work on the logical foundations of mathematics written by Bertrand Russel and Alfred Whitehead between 1910 and 1913. Its goal was to show that all of the concepts of mathematics can be expressed in logic, and that all of the theorems of mathematics can be proved using only the logical axioms and rules of inference of logic. It covered set theory, ordinal numbers and real numbers, and it showed in principle that large parts of mathematics could be developed using *logicism*.

It avoided the problems with contradictions that arose with Frege's system by introducing Russell's theory of types in the system. The theory of types meant that one could no longer speak of the set of all sets, as a set of elements is of a different type from that of each of its elements, and so Russell's paradox was avoided. It remained an open question at the time as to whether the *Principia* was consistent and complete. However, it was clear from the three-volume work that the development of mathematics using the approach of the *Principia* was extremely lengthy and time consuming.

11.3   Decidability

The question remained whether these axioms and rules of inference are sufficient to decide any mathematical question that can be expressed in these systems. Hilbert believed that every mathematical problem could be solved, and that the truth or falsity of any mathematical proposition could be determined in a finite number of steps. He outlined 23 key problems in 1900 that needed to be solved by mathematicians in the twentieth century.

He believed that the formalism of mathematics would allow a mechanical procedure (or algorithm) to determine whether a particular statement was true or false. The problem of the decidability of mathematics is known as the decision problem (*Entscheidungsproblem*).

The question of the decidability of mathematics had been considered by Leibnitz in the seventeenth century. He had constructed a mechanical calculating machine, and wondered if a machine could be built that could determine whether particular mathematical statements are true or false.

Definition 11.1 (Decidability) *Mathematics is decidable if the truth or falisity of any mathematical proposition may be determined by an algorithm.*

Church and Turing independently showed this to be impossible in 1936. Turing showed that decidability was related to the halting problem for Turing machines, and that if first-order logic were decidable then the halting problem for Turing machines could be solved. However, he had already proved that there was no general algorithm to determine whether an abritary Turing machine halts. Therefore, first order logic is undecidable.

Fig. 11.2 Kurt Gödel

The question as to whether an arbitrary Turing machine halts or not can be formulated as a first-order statement. If a general decision procedure exists for the first-order logic, then the statement of whether an arbitrary Turing machine halts or not is within the scope of the decision algorithm. However, Turing had already proved that the halting problem for Turing machines is not computable, i.e. it is not possible algorithmically to decide whether or not an arbitrary Turing machine will halt or not. Therefore, since there is no general algorithm that can decide whether any given Turing machine halts, there is no general decision procedure for the first-order logic. The only way to determine whether a statement is true or false is to try to solve it. However, if one tries but does not succeed, this does not prove that an answer does not exist.

There are first-order theories that are decidable. However, the first-order logic that includes Peano's axioms of arithmetic (or any formal system that includes addition and multiplication) cannot be decided by an algorithm. Propositional logic is decidable as there is a procedure (e.g., using a truth table) to determine if an arbitrary formula is valid in the calculus.

A well-formed formula is valid if it follows from the axioms of the first-order logic. A formula is valid if and only if it is true in every interpretation of the formula in the model. Gödel proved that the first order predicate calculus is *complete*, i.e. all truths in the predicate calculus can be proved in the language of the calculus.

Definition 11.2 (Completeness) *A formal system is complete if all the truths in the system can be derived from the axioms and rules of inference.*

Gödel's *first incompleteness theorem* showed that the first-order arithmetic is incomplete; i.e. there are truths in the first-order arithmetic that cannot be proved in the language of the axiomatisation of first-order arithmetic. Gödel's *second incompleteness theorem* showed that that any formal system extending basic arithmetic cannot prove its own consistency within the formal system (Fig. 11.2).

Definition 11.3 (Consistency) *A formal system is consistent if there is no formula A such that A and ¬A are provable in the system (i.e. there are no contradictions in the system).*

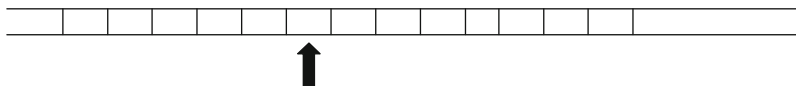**Fig. 11.3** Potentially infinite tape

11.4   Computability

Alonzo Church developed the lambda calculus (discussed briefly in Chap. 10) in the mid-1930s, as part of his work into the foundations of mathematics. Turing published a key paper on computability in 1936, which introduced the theoretical machine known as the Turing machine. This machine is computationally equivalent to the lambda calculus, and is capable of performing any conceivable mathematical problem that has an algorithm. The term '*algorithm*' is named after the Persian mathematician Al-Khwarizmi.

Definition 11.4 (Algorithm) *An algorithm* (*or effective procedure*) *is a finite set of unambiguous instructions to perform a specific task*.

A function is *computable* if there is an effective procedure or algorithm to compute *f* for each value of its domain. The algorithm is finite in length and sufficiently detailed so that a person can execute the instructions in the algorithm. The execution of the algorithm will halt in a finite number of steps to produce the value of $f(x)$ for all x in the domain of *f*. However, if x is not in the domain of *f* then the algorithm may produce an answer saying so, or it may get stuck, or it may run forever, never halting.

The *Church–Turing Thesis* that states that *any computable function may be computed by a Turing machine*. There is overwhelming evidence in support in support of this thesis, including the fact that alternative formalisations of computability in terms of lambda calculus, recursive function theory, and Post systems have all been shown to be equivalent to the Turing machines.

A Turing machine consists of a head and a potentially infinite tape that is divided into cells. Each cell on the tape may be either blank or printed with a symbol from a finite alphabet of symbols. The input tape may initially be blank or have a finite number of cells containing symbols. At any step, the head can read the contents of a frame. The head may erase a symbol on the tape, leave it unchanged, or replace it with another symbol. It may then move one position to the right, one position to the left, or not at all. If the frame is blank, the head can either leave the frame blank or print one of the symbols.

Turing believed that a human being with finite equipment and with an unlimited supply of paper could do every calculation. The unlimited supply of paper is formalized in the Turing machine by a tape marked off in cells (Fig. 11.3).

Definition 11.5 (Turing Machine) *A Turing machine* $M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$ *is a 7-tuple* and defined formally in [HoU:79] where:

- *Q* is a finite set of *states*
- Γ is a finite set of the *tape alphabet/symbols*

- $b \in \Gamma$ is the *blank symbol*. (This is the only symbol that is allowed to occur infinitely often on the tape during each step of the computation.)
- Σ is the set of input symbols and is a subset of Γ (i.e. $\Gamma = \Sigma \cup \{b\}$).
- $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function. This is a partial function where L is the left shift, R is the right shift.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final or accepting states.

The Turing machine is simple theoretical machine, but it is equivalent to an actual physical computer in the sense that they both compute exactly the same set of functions. A Turing machine is easier to analyse and prove things about than a real computer. However, Turing machines are not suitable for programming, and they do not provide a good basis for studying programming or programming languages.

A Turing machine is essentially a finite state machine (FSM) with an unbounded tape. The machine may read from and write to the tape and the tape provides memory and acts as the store. The finite state machine is essentially the control unit of the machine, whereas the tape is a potentially infinite and unbounded store. A real computer has a large but finite store whereas the store in a Turing machine is potentially infinite. However, the store in a real computer may be extended with backing tapes and disks, and in a sense may be regarded as unbounded. The maximum amount of tape that may be read or written within n steps is n.

A Turing machine has an associated set of rules that defines its behaviour. These rules are defined by the transition function that specify the actions that a machine will perform with respect to a particular input. The behaviour will depend on the current state of the machine and the contents of the tape.

A Turing machine may be programmed to solve any problem for which there is an algorithm. However, if the problem is unsolvable, then the machine will either stop in a non-accepting state or compute forever. The solvability of a problem may not be determined beforehand, but, there is, of course, some answer (i.e. either the machine either halts or it computes forever).

Turing showed that there was no solution to the decision problem (*Entscheidungsproblem*) posed by Hilbert. Hilbert believed that the truth or falsity of a mathematical problem could always be determined by a mechanical procedure, and he believed that the first-order logic is decidable, i.e. there is a decision procedure to determine if an arbitrary formula is a theorem of the logical system.

Turing was skeptical on the decidability of first-order logic. The Turing machine played a key role in refuting Hilbert's claim of its decidability.

Turing also introduced the concept of a Universal Turing Machine which is able to simulate any other Turing machine. His results on computability were proved independently of Church's lambda calculus equivalent results in computability. He studied at Princeton University in 1937 and 1938 and was awarded a Ph.D. in 1938. His research supervisor was Alonzo Church.[3]

---

[3] Alonzo Church was a famous American mathematician and logician who developed the lambda calculus. He also showed that Peano arithmetic and the first-order logic were undecidable. Lambda calculus is equivalent to Turing machines, and whatever may be computed is computable by Lambda calculus or a Turing machine.

Question 11.1 (Halting Problem) *Given an arbitrary program is there an algorithm to decide whether the program will finish running or will continue running forever? Another words, given a program and an input will the program eventually halt and produce an output or will it run forever?*

Note (Halting Problem)   The halting problem was one of the first problems that was shown to be undecidable: i.e., there is no general decision procedure or algorithm that may be applied to an arbitrary program and input to decide whether the program halts or not when run with that input.

Proof   We assume that there is an algorithm (i.e a computable function function $H(i,j)$) that takes any program i (program i refers to the ith program in the enumeration of all the programs) and arbitrary input j to the program such that:

$$H(i, j) = \begin{cases} 1 & \text{If program } i \text{ halts on input } j. \\ 0 & \text{otherwise} \end{cases}$$

We then employ a diagonalisation argument[4] to show that every computable total function f with two arguments differs from the desired function H. First, we construct a partial function g from any computable function f with two arguments such that g is computable by some program e.

$$g(i) = \begin{cases} 0 & \text{if } f(i,i) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

There is a program e that computes g and this program is one of the programs in which the halting problem is defined. One of the following two cases must hold:

$$g(e) = f(e, e) = 0 \tag{11.1}$$

In this case $H(e,e) = 1$ because e halts on input e.

$$g(e) \text{ is undefined and } f(e, e) \neq 0. \tag{11.2}$$

In this case $H(e,e) = 0$ because the program e does not halt on input e.

In either case, f is not the same function as H. Further, since f was an arbitrary total computable function all such functions must differ from H. Hence, the function H is not computable and there is no such algorithm to determine whether an arbitrary Turing machine halts for an input x. Therefore, the halting problem is not decidable.

---

[4] This is similar to Cantor's diagonalisation argument that shows that the Real numbers are un-countable. This argument assumes that it is possible to enumerate all real numbers between 0 and 1, and it then constructs a number whose nth decimal differs from the nth decimal position in the nth number in the enumeration. If this holds for all n, then the newly defined number is not among the enumerated numbers.

11.5   Computational Complexity

An algorithm is of little practical use if it takes millions of years to compute particular instances. There is a need to consider the efficiency of the algorithm due to practical considerations. Chapter 8 discussed cryptography and the RSA algorithm. The security of the RSA encryption algorithm is due to the fact that there is no known efficient algorithm to determine the prime factors of a large number.

There are often slow and fast algorithms for the same problem, and a measure of complexity is the number of steps in a computation. An algorithm is of *time complexity* $f(n)$ if for all n and and all inputs of length n the execution of the algorithm takes at most $f(n)$ steps.

An algorithm is said to be *polynomially bounded* if there is a polynomial $p(n)$ such that for all n and and all inputs of length n the execution of the algorithm takes at most $p(n)$ steps. The notation P is used for all problems that can be solved in polynomial time.

A problem is said to be *computationally intractable* if it is not in P. That is, there is no known algorithm in polynomial time for the problem.

A problem L is said to be in the set *NP* (non-deterministic polynomial time problems) if any given solution to L can be verified quickly in polynomial time. A non-deterministic Turing machine may have several possibilities for its behaviour, and an input may give rise to several computations.

A problem is *NP complete* if it is in the set NP of non-deterministic polynomial time problems and it is also in the class of *NP hard* problems. A key characteristic to NP complete problems is that there is no known fast solution to them, and the time required to solve the problem using known algorithms increases quickly as the size of the problem grows. Often, the time required to solve the problem is in billions or trillions of years. Although any given solution can be verified quickly, there is no known efficient way to find a solution.

11.6   Review Questions

1. Explain computability and Decidability.
2. What were the goals of formalism and how successful was this movement in mathematics?
3. What is a formal system?
4. Explain the difference between consistency, completeness and decidability.
5. Describe a Turing maching and explain its significance in computability.
6. Describe the halting problem and show that it is undecidable.
7. Discuss the complexity of an algorithitm and explain terms such as 'polynomial bounded', 'computationally intractable', and 'NP complete'.

11.7 Summary

A function f is computable if there exists an algorithm that produces the value of f correctly for each possible argument of f. The computation of f for a particular argument x just involves following the instructions in the algorithm, and it produces the result $f(x)$ in a finite number of steps if x is in the domain of f.

The concept of computability may be made precise in several equivalent ways such as Church's lambda calculus, recursive function theory or by the theoretical Turing machines. The Turing machine is a mathematical machine with a potentially infinite tape divided into frames (or cells) in which very basic operations can be carried out. The set of functions that are computable are those that are computable by a Turing machine.

A formal system contains meaningless symbols together with rules for manipulating them, and is generally intended to represent some aspect of the real world. The individual formulas are certain finite sequences of symbols obeying the syntactic rules of the formal language. A formal system consists of a formal language, a set of axioms and rules of inference.

Church and Turing independently showed in 1936 that mathematics is not decidable. In other words, it is not possible to determine the truth or falisty of any mathematical proposition by an algorithm. Turing had already proved that the halting problem for Turing machines is not computable, and he applied this result to first order to show that the first-order logic is undecidable. That is, the only way to determine whether a statement is true or false is to try to solve it.

An algorithm is of little practical use if it takes millions of years to compute the solution. There is a need to consider the efficiency of the algorithm due to practical considerations. The class of polynomial time bound problems and non-deterministic polynomial time problems were considered, and it was noted that the security of various cryptographic algorithms is due to the fact that there are no time efficient algorithms to determine the prime factors of large integers.

A detailed account of decidability and computability is given in [RoS:94].

Chapter 12
Probability, Statistics and Software Reliability

Key Topics
Sample Spaces
Random Variables
Mean, Mode and Median
Variance
Normal Distributions
Histograms
Hypothesis Testing
Software Reliability Models

12.1 Introduction

Statistics is an empirical science that is concerned with the collection, organisation, analysis, interpretation and presentation of data. The data collection needs to be planned and this may include surveys and experiments. Statistics is widely used by government and industrial organisations, and it is employed for forecasting as well as for presenting trends. They allow the behaviour of a population to be studied and inferences to be made about the population. These inferences may be tested (*hypothesis testing*) to ensure their validity.

The analysis of statistical data allows an organisation to understand its performance in key areas and to identify problematic areas. Organisations will often examine performance trends over time and will devise appropriate plans and actions to address problematic areas. The effectiveness of the actions taken will be judged by improvements in performance trends over time.

It is often not possible to study the entire population, and instead, a representative subset or sample of the population is chosen. This *random sample* is used to make inferences regarding the entire population, and it is essential that the sample chosen is indeed random and representative of the entire population. Otherwise, the inferences made regarding the entire population will be invalid.

A statistical experiment is a causality study that aims to draw a conclusion regarding the values of a *predictor variable*(s) on a *response variable*(s). For example, a statistical experiment in the medical field may be conducted to determine if there is a causal relationship between the use of a particular drug and the treatment of a medical condition such as lowering of cholesterol in the population. A statistical experiment involves:

- Planning the research
- Designing the experiment
- Performing the experiment
- Analysing the results
- Presenting the results

Probability is a way of expressing the likelihood of a particular event occurring. It is normal to distinguish between the frequency interpretation and the subjective interpretation of probability. For example, if a geologist sates that "there is a 70 % chance of finding gas in a certain region" then this statement is usually interpreted in two ways:

- The geologist is of the view that over the long run, 70 % of the regions whose environment conditions are very similar to the region under consideration have gas (*Frequency Interpretation*).
- The geologist is of the view that it is likely that the region contains gas, and that 0.7 is a measure of the geologist's belief in this hypothesis. (*Personal Interpretation*)

12.2   Probability Theory

Probability theory provides a mathematical indication of the likelihood of an event occurring, and the probability is between 0 and 1. A probability of 0 indicates that the event cannot occur whereas a probability of 1 indicates that the event is guaranteed to occur. If the probability of an event is greater than 0.5, then this indicates that the event is more likely to occur than not.

A *sample space* is the set of all possible outcomes of an experiment, and an *event* E is a subset of the sample space. For example, the sample space for the experiment of tossing a coin is the set of all possible outcomes of this experiment, i.e. head or tails. The event that the toss results a tail is a subset of the sample space.

$$S = \{h, t\} \quad E = \{t\}$$

Similarly, the sample space for the gender of a newborn baby is the set of outcomes: i.e. the newborn baby is a boy or a girl. The event that the baby is a girl is a subset of the sample space.

$$S = \{b, g\} \quad E = \{g\}$$

For any two events E and F of a sample space S we can also consider the union and intersection of these events. That is,

- E ∪ F consists of all outcomes that are in E or F or both.
- E ∩ F (normally written as EF) consists of all outcomes that are in both E and F.
- E^c denotes the complement of E with respect to S and represents the outcomes of S that are not in E.

If EF = ∅ then there are no outcomes in both E and F, and so the two events E and F are mutually exclusive. The union and intersection of two events can be extended to the union and intersection of a family of events $E_1, E_2, \cdots E_n$ (i.e., $\cup_{i=1}^{n} E_i$ and $\cap_{i=1}^{n} E_i$).

12.2.1   Laws of Probability

The laws of probability essentially state that the probability of an event is between 0 and 1, and the union of a mutually disjoint set of events is the sum of their individual probabilities.

1. P(S) = 1
2. P(∅) = 0.
3. $0 \leq P(E) \leq 1$
4. For any sequence of mutually exclusive events $E_1, E_2, \cdots E_n$. (i.e. $E_i\ E_j = ∅$ where $i \neq j$) then the probability of the union of these events is the sum of their individual probabilities, i.e.

$$P(\cup_{i=1}^{n} E_i) = \sum_{i=1}^{n} P(E_i).$$

The probability of the union of two events (not necessarily disjoint) is given by:

$$P(E \cup F) = P(E) + P(F) - P(EF)$$

The probability of an event E not occurring is denoted by E^c and is given by $1 - P(E)$. The probability of an event E occurring given that an event F has occurred is termed the *conditional probability* (denoted by $P(E|F)$) and is given by:

$$P(E|F) = \frac{P(EF)}{P(F)} \quad \text{where } P(F) > 0$$

This formula allows us to deduce that:

$$P(EF) = P(E|F)P(F)$$

Bayes formula enables the probability of an event E to be determined by a weighted average of the conditional probability of E given that the event F occurred and the conditional probability of E given that F has not occurred:

$$E = E \cap S = E \cap (F \cup F^c)$$
$$= EF \cup EF^c$$
$$P(E) = P(EF) + P(EF^c) \quad (\text{since } EF \cap EF^c = \varnothing)$$
$$= P(E|F) P(F) + P(E|F^c)P(F^c)$$
$$= P(E|F)P(F) + P(E|F^c) (1 - P(F))$$

Two events E, F are *independent* if knowledge that F has occurred does not change the probability that E has occurred. That is, $P(E|F) = P(E)$ and since $P(E|F) = P(EF)/P(F)$ we have that two events E and F are independent if:

$$P(EF) = P(E)P(F)$$

Two events E and F that are not independent are said to be *dependent*.

12.2.2   Random Variables

Often, some numerical quantity determined by the result of the experiment is of interest rather than the result of the experiment itself. These numerical quantities are termed *random variables*. A random variable is termed *discrete* if it can take on a finite or countable number of values, otherwise it is termed *continuous*.

The *distribution function* of a random variable is the probability that the random variable X takes on a value less than or equal to *x*. It is given by:

$$F(x) = P\{X \le x\}$$

All probability questions about X can be answered in terms of its distribution function F. For example, the computation of $P\{a < X < b\}$ is given by:

$$P\{a < X < b\} = P\{X \le b\} - P\{X \le a\}$$
$$= F(b) - F(a)$$

The *probability mass function* for a discrete random variable X (denoted by $p(a)$) is the probability that it is a certain value. It is given by:

$$p(a) = P\{X = a\}$$

Further, $F(a)$ can also be expressed in terms of the probability mass function

$$F(a) = \sum_{\forall x \le a} p(x)$$

X is a continuous random variable if there exists a non-negative function $f(x)$ (termed the *probability density function*) defined for all $x \in (-\infty, \infty)$ such that

$$P\{X \in B\} = \int_B f(x)dx$$

All probability statements about X can be answered in terms of its density function $f(x)$. For example:

$$P\{a \leq X \leq b\} = \int_a^b f(x)dx$$

$$P\{X \in (-\infty, \infty)\} = 1 = \int_{-\infty}^{\infty} f(x)dx$$

The function $f(x)$ is termed the probability density function and the probability distribution function $F(a)$ is defined by:

$$F(a) = P\{X \leq a\} = \int_{-\infty}^{a} f(x)dx$$

Further, the first derivative of the probability distribution function yields the probability density function. That is,

$$\frac{d}{da}F(a) = f(a).$$

The expected value (i.e. the *mean*) of a discrete random variable X (denoted E[X]) is given by the weighted average of the possible values of X:

$$E[X] = \begin{cases} \sum_i x_i P\{X = x_i\} & \text{Discrete Random variable} \\ \int_{-\infty}^{\infty} xf(x)dx & \text{Continuous Random variable} \end{cases}$$

Further, the expected value of a function of a random variable is given by E[g(X)] and is defined for the discrete and continuous case respectively.

$$E[g(X)] = \begin{cases} \sum_i g(x_i)P\{X = x_i\} & \text{Discrete Random variable} \\ \int_{-\infty}^{\infty} g(x)f(x)dx & \text{Continuous Random variable} \end{cases}$$

The *variance* of a random variable is a measure of the spread of values from the mean, and is defined by:

$$\text{Var}(X) = E[X^2] - (E[X])^2$$

The standard deviation σ is given by the square root of the variance, that is,

$$\sigma = \sqrt{\text{Var}(X)}$$

The *covariance* of two random variables is a measure of the relationship between two random variables X and Y and indicates the extent to which they both change (in either similar or opposite ways) together. It is defined by:

$$\text{Cov}(X, Y) = E[XY] - E[X] \cdot E[Y].$$

It follows that the covariance of two independent random variables is zero. Variance is a special case of covariance (when the two random variables are identical). This follows since $\text{Cov}(X, X) = E[X.X] - (E[X])(E[X]) = E[X^2] - (E[X])^2 = \text{Var}(X)$.

A positive covariance ($\text{Cov}(X, Y) \geq 0$) indicates that Y tends to increase as X does, whereas a negative covariance indicates that Y tends to decrease as X increases.

The *correlation* of two random variables is an indication of the relationship between two variables X and Y. If the correlation is negative then Y tends to decrease as X increases and if it is positive number, then Y tends to increase as X increases. The correlation coefficient is a value that is between ± 1 and it is defined by:

$$\text{Corr}(X, Y) = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}$$

Once the correlation between two variables has been calculated, the probability that the observed correlation was due to chance can be computed. This is to ensure that the observed correlation is a real one and not due to a chance occurrence.

There are a number of special random variables, and these include the Bernouilli trial, where there are just two possible outcomes of an experiment, i.e. success or failure. The probability of success and failure is given by:

$$P\{X = 0\} = 1 - p$$
$$P\{X = 1\} = p$$

The mean of the Bernouilli distribution is given by p and the variance by $p(1-p)$. The *Binomial distribution* involves n Bernouilli trials, each of which results in success or failure. The probability of i successes from n trials is then given by:

$$P\{X = i\} = (^n_i)p^i(1 - p)^{n-i}$$

with the mean of the Binomial distribution given by np, and the variance is given by $np(1 - p)$.

The *Poisson distribution* may be used as an approximation to the Binomial distribution when n is large and p is small. The probability of i successes is given by:

$$P\{X = i\} = \frac{e^{-\lambda}\lambda^i}{i!}$$

and the mean and variance of the Poisson distribution is given by λ.

There are many other well-known distributions such as the *hypergeometric distribution* that describes the probability of i successes in n draws from a finite population without replacement; the *uniform distribution*; the *exponential distribution*, the *normal distribution* and the *gamma* distribution.

The mean and variance of these distributions are summarised in Table 12.1.

Table 12.1 Probability distributions. (Source: [Ros:87])

Distribution name	Density function	Mean and variance
Hypergeometric	$P\{X = i\} = \binom{N}{i}\binom{M}{n-i}/\binom{N}{n}^{+M}$	$nN/N + M$, $np(1 - p)[1 - (n - 1)/N + M - 1]$
Uniform	$f(x) = 1/(\beta - \alpha)\alpha \le x \le \beta, 0$	$(\alpha + \beta)/2$, $(\beta - \alpha)^2/12$
Exponential	$f(x) = \lambda e^{-\lambda x}$	$1/\lambda$, $1/\lambda^2$
Normal	$f(x) = 1/\sqrt{2\pi}\sigma[e^{-(x-\mu)^2/2\sigma^2}]$	μ, σ^2
Gamma	$f(x) = \lambda e^{-\lambda x}(\lambda x)^{\alpha-1}/\Gamma(\alpha)$	α/λ, α/λ^2

12.3   Statistics

The field of statistics is concerned with summarising, digesting and extracting information from large quantities of data. It provides a collection of methods for planning an experiment, and analyzing data to draw accurate conclusions from the experiment. We distinguish between descriptive statistics and inferential statistics.

Descriptive Statistics This is concerned with describing the information in a set of data elements in graphical format, or by describing its distribution.

Inferential Statistics This is concerned with making inferences with respect to the population by using information gathered in the sample.

12.3.1   Abuse of Statistics

Statistics are extremely useful in drawing conclusions about a population. However, it is essential that the random sample chosen is actually random, and that the experiment is properly conducted to ensure valid conclusions to be are inferred. Some examples of the abuse of statistics include:

- The sample size may be too small to draw conclusions
- It may not be a genuine random sample of the population.
- Graphs may be drawn to exaggerate small differences
- Area may be misused in representing proportions.
- Misleading percentages may be used.

The quantitative data used in statistics may be discrete or continuous. *Discrete data* is numerical data that has a finite number of possible values, and *continuous data* is numerical data that has an infinite number of possible values.

12.3.2   Statistical Sampling

Statistical sampling is concerned with the methodology of choosing a random sample of a population, and the study of the sample with the goal of drawing valid conclusions

Table 12.2 Sampling techniques

Sampling technique	Description
Systematic	Every kth member of the population is sampled
Stratified	The population is divided into two or more strata and each subpopulation (stratum) is then sampled. Each element in the subpopulation shares the same characteristics (e.g. age groups, gender)
Cluster	A population is divided into clusters and a few of these clusters are exhaustively sampled (i.e. every element in the cluster is considered). This approach is often used by government organisations
Convenience	Sampling is done as convenient and often allows the element to choose whether or not it is sampled

Table 12.3 Types of survey

Survey type	Description
Direct measurement	This may involve a direct measurement of all in the sample (e.g. the height of students in a class)
Mail Survey	This involves sending a mail survey to the sample. This may have a lower response rate and may thereby invalidate the findings
Phone Survey	This is a reasonably efficient and cost effective way to gather data. However, refusals or hang-ups may affect the outcome
Personal interview	This tends to be expensive and time consuming, but it allows detailed information to be collected
Observational study	An observational study allows individuals to be studied, and the variables of interest to be measured
Experiment	An experiment imposes some treatment on individuals in order to study the response

about the entire population. The assumption is that if a genuine representative sample of the population is chosen, then a detailed study of the sample will provide insight into the whole population. This helps to avoid a lengthy expensive (and potentially infeasible) study of the entire population.

The sample chosen must be random and the sample size sufficiently large to enable valid conclusions to be made for the entire population.

Random Sample   A *random sample* is a sample of the population such that each member of the population has an equal chance of being chosen.

There are various ways of generating a random sample from the population including (Table 12.2).

Once the sample is chosen the next step is to obtain the required information from the sample. This may be done by interviewing each member in the sample; phoning each member; conducting a mail survey, and so on (Table 12.3).

12.3.3   Averages in a Sample

The term 'average' generally refers to the arithmetic *mean* of a sample, but it may also refer to the statistical *mode* or *median* of the sample. These terms are defined below:

Mean The *arithmetic mean* of a set of n numbers is defined to be the sum of the numbers divided by n. That is, the arithmetic mean for a sample of size n is given by:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

The actual mean of the population is denoted by μ, and it may differ from the sample mean.

Mode The mode is the data element that occurs most freqently in the sample. It is possible that two elements occur with the same frequency, and if this is the case then we are dealing with a bi-modal or possibly a multi-modal sample.

Median The median is the middle element when the data set is arranged in increasing order of magnitude.

   If there are an odd number of elements in the sample the median is the middle element. Otherwise, the median is the arithmetic mean of the two middle elements.

Mid range The midrange is the arithmetic mean of the highest and lowest data elements in the sample, that is, $(x_{max} + x_{min})/2$.

   The arithmetic mean is the most widely used average in statistics.

12.3.4   *Variance and Standard Deviation*

An important characteristic of a sample is its distribution and the spread of each element from some measure of central tendency (e.g. the mean). One elementary measure of dispersion is that of the sample *range*, and it is defined to be the difference between the maximum and minimum value in the sample. That is, the sample range is defined to be:

$$\text{range} = x_{max} - x_{min}.$$

The sample range is not a reliable measure of dispersion as only two elements in the sample are used, and extreme values in the sample can distort the range to be very large even if most of the elements are quite close to one another.

   The standard deviation is the most common way to measure dispersion, and it gives the average distance of each element in the sample from the mean. The sample standard deviation is denoted by s and is defined by:

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}}$$

The population standard deviation is denoted by σ and is defined by:

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Fig. 12.1 Carl Friedrich
Gauss

Variance is another measure of dispersion and it is defined as the square of the standard deviation. The sample variance is given by:

$$s^2 = \frac{\sum (x_i - \bar{x})^2}{n - 1}$$

The population variance is given by:

$$\sigma^2 = \frac{\sum (x_i - \mu)^2}{N}$$

12.3.5 Bell-shaped (Normal) Distribution

The German mathematician, Gauss (Fig. 12.1) originally studied the normal distribution (also known as the Gaussian distribution). The distribution is shaped like a bell and so is popularly known as the bell-shaped distribution. The empirical frequencies of many natural populations exhibit a bell-shaped (normal) curve.

The *normal distribution N* has mean μ, and standard deviation σ. Its density function $f(x)$ is given by:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

Where $-\infty < x < \infty$.

The *unit* (or *standard*) normal distribution Z(0, 1) has mean 0 and standard deviation of 1. Every normal distribution may be converted to the unit normal distribution by $Z = (X - \mu)/\sigma$, and every probability statement about X has an equivalent probability statement about Z. The unit normal density function is given by:

$$f(y) = \frac{1}{\sqrt{2\pi}} e^{-y^2/2}$$

For a normal distribution 68.2 % of the data elements lie within one standard deviation of the mean; 95.4 % of the population lies within two standard deviations of the mean,

and 99.7 % of the data lies within three standard deviations of the mean. For example, the shaded area under the curve within two standard deviations of the mean represents 95 % of the population (Fig. 12.2).

A fundamental result in probability theory is the *Central Limit Theorem*, and this theorem essentially states that the sum of a large number of independent and identically distributed random variables has a distribution that is approximately normal. That is, suppose $X_1, X_2, \cdots X_n$ is a sequence of independent random variables each with mean μ and variance σ^2. Then for large n the distribution of

$$\frac{X_1 + X_2 + \cdots + X_n - n\mu}{\sigma\sqrt{n}}$$

is approximately that of a unit normal variable Z. One application of the central limit theorem is in relation to the binomial random variables, where a binomial random variable with parameters (n, p) represents the number of successes of n independent trials, where each trial has a probability of p of success. This may be expressed as:

$$X = X_1 + X_2 + \cdots + X_n$$

where $X_i = 1$ if the ith trial is a success and is 0 otherwise. $E(X_i) = p$ and $Var(X_i) = p(1 - p)$, and then by applying the central limit theorem it follows that for large n

$$\frac{X - np}{\sqrt{np(1 - p)}}$$

will be approximately a unit normal variable (which becomes more normal as n becomes larger).

The sum of independent normal random variables is normally distributed. It can be shown that the sample average of $X_1, X_2, \cdots X_n$ is normal, with a mean equal to the population mean but with a variance reduced by a factor of $1/n$.

$$E(\bar{X}) = \sum_{i=1}^{n} \frac{E(X_i)}{n} = \mu$$

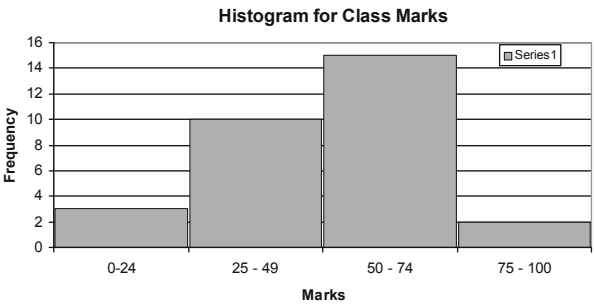$$Var(\bar{X}) = \frac{1}{n^2} \sum_{i=1}^{n} Var(X_i) = \frac{\sigma^2}{n}$$

Table 12.4 Frequency
table—Salary

Profession	Salary	Frequency
Project manager	65,000	3
Architect	65,000	1
Programmer	50,000	8
Tester	45,000	2
Director	90,000	1

Table 12.5 Frequency
table—Test results

Mark	Frequency
0–24	3
25–49	10
50–74	15
75–100	2

Fig. 12.3 Histogram test
results

It follows from this that the following is a unit normal random variable.

$$\sqrt{n}\frac{(X - \mu)}{\sigma}$$

The term *six-sigma* (6σ) is a methodology concerned with continuous process im-provement and aims for very high quality close to perfection. A 6σ process is one in which 99.9996 % of the products are expected to be free from defects (3.4 defects per million).

12.3.6 *Frequency Tables, Histograms and Pie Charts*

A frequency table is used to present or summarise data. It lists the data classes (or categories) in one column and the frequency of the category in another column (Tables 12.4 and 12.5).

A histogram is a way to represent data in bar chart format. The data is divided into intervals where an interval is a certain range of values. The horizontal axis of the histogram contains the intervals (also known as buckets) and the vertical axis shows the frequency (or relative frequency) of each interval. The bars represent the frequency and there is no space between the bars (Fig. 12.3).

Fig. 12.4 Pie chart test results
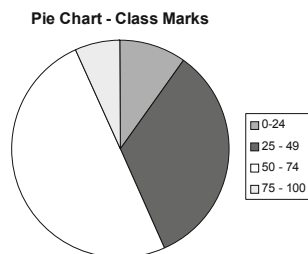
A histogram has an associated shape. For example, it may resemble a normal distribution, a bi-modal or multi-modal distribution. It may be positively or negatively skewed. The construction of a histogram first involves the construction of a frequency table where the data is divided into disjoint classes and the frequency of each class is determined (Fig. 12.4).

A pie chart offers an alternate way to histograms in the presentation of data. A frequency table is first constructed, and the pie chart presents a visual representation of the percentage in each data class (Fig. 12.4).

12.3.7   Hypothesis Testing

The basic concept of inferential statistics is *hypothesis testing*, where a hypothesis is a statement about a particular population whose truth or falsity is unknown. Hypothesis testing is concerned with determining whether the values of the random sample from the popularion are consistent with the hypothesis. There are two mutually exclusive hypotheses: one of these is the null hypothesis H_0 and the other is the alternate research hypothesis H_1. The null hypothesis H_0 is what the researcher is hoping to reject, and the research hypothesis H_1 is what the researcher is hoping to accept.

Statistical testing is then employed to test the hypothesis, and the result of the test is that we either reject the null hypothesis (and therefore accept the alternative hypothesis), or that we fail to reject it (i.e. we accept) the null hypothesis. The rejection of the null hypothesis means that the null hypothesis is highly unlikely to be true, and that the research hypothesis should be accepted.

Statistical tesing is conducted at a certain level of significance, with the probability of the null hypothesis H_0 being rejected when it is true never greater than α. The value α is called the level of significance of the test, with α usually being 0.1, 0.05 or 0.005. A significance level β may also be applied to with respect to accepting the null hypothesis H_0 when H_0 is false, and usually $\alpha = \beta$.

The objective of a statistical test is not to determine whether or not H_0 is actually true, but rather to determine whether its validity is consistent with the observed data. That is, H_0 should only be rejected if the resultant data is very unlikely if H_0 is true (Table 12.6).

The errors that can occur with hypothesis testing include type 1 and type 2 errors. Type 1 errors occur when we reject the null hypothesis when the null hypothesis is

Table 12.6 Hypothesis testing

Action	H_0 true, H_1 false	H_0 false, H_1 true
Reject H_1	Correct	False Positive—Type 2 error $P(\text{Accept } H_0 | H_0 \text{ false}) = \beta$
Reject H_0	False Negative—Type 1 error $P(\text{Reject } H_0 | H_0 \text{ true}) = \alpha$	Correct

actually true. Type 2 errors occur when we accept the null hypothesis when the null hypothesis is false.

For example, an example of a false positive is where the results of a blood test comes back positive to indicate that a person has a particular disease when in fact the person does not have the disease. Similarly, an example of a false negative is where a blood test is negative indicating that a person does not have a particular disease when in fact the person does. Both errors can potentially be very serious.

The terms α and β represent the level of significance that will be accepted, and normally $\alpha = \beta$. In other words, α is the probability that we will reject the null hypothesis when the null hypothesis is true, and β is the probability that we will accept the null hypothesis when the null hypothesis is false.

Testing a hypothesis at the $\alpha = 0.05$ level is equivalent to establishing a 95 % confidence interval. For 99 % confidence α will be 0.01, and for 99.999 % confidence then α will be 0.00001.

The hypothesis may be concerned with testing a specific statement about the value of an unknown parameter θ of the population. This test is to be done at a certain level of significance, and the unknown parameter may, for example, be the mean or variance of the population. An estimator for the unknown parameter is determined, and the hypothesis that this is an accurate estimate is rejected if the random sample is not consistent with it. Otherwise, it is accepted.

The steps involved in hypothesis testing include:

1. Establish the null and alternative hypothesis
2. Establish error levels (significance)
3. Compute the test statistics (often a *t*-test)
4. Decide on whether to accept or reject the null hypothesis.

The difference between the observed and expected test statistic, and whether the difference could be accounted for by normal sampling fluctuations is the key to the acceptance or rejection of the null hypothesis.

12.4   Software Reliability

The design and development of high-quality software has become increasingly important for society. Many software companies desire a sound mechanism to predict the reliability of their software prior to its deployment at the customer site, and this has led to a growing interest in software reliability models.

Definition 12.1 (Software Reliability)   *Software reliability* is defined as the probability that the program works without failure for a specified length of time, and is a statement of the future behaviour of the software. It is generally expressed in terms of the *mean-time-to-failure* (MTTF) or the *mean-time-between-failure* (MTBF).

Statistical sampling techniques are often employed to predict the reliability of hardware, as it is not feasible to test all items in a production environment. The quality of the sample is then used to make inferences on the quality of the entire population, and this approach is effective in manufacturing environments where variations in the manufacturing process often lead to defects in the physical products.

There are similarities and differences between hardware and software reliability. A hardware failure may arise due to a component wearing out due to its age, and often a replacement is required. Most hardware components are expected to last for a certain period of time, and the variation in the failure rate of a hardware component are often due to the manufacturing process and to the operating environment of the component. Good hardware reliability predictor models have been developed, and each hardware component has an expected mean time to failure. The reliability of a product may be determined from the reliability of the individual components of the hardware.

Software is an intellectual undertaking involving a team of designers and programmers. It does not physically wear out and software failures manifest themselves from particular user inputs. Each copy of the software code is identical and the software is either correct or incorrect, that is, software failures are due to design and implementation errors rather than physical wearing out. *The software community has not yet developed a sound software reliability predictor model.*

The software population to be sampled consists of all possible execution paths of the software, and since this is potentially infinite it is generally not possible to perform exhaustive testing.

The way in which the software is used (i.e. the inputs entered by the users) will impact upon its perceived reliability. Let I_f represent the fault set of inputs (i.e. $i_f \in I_f$ if and only if the input of i_f by the user leads to failure). The randomness of the time to software failure is due to the unpredictability in the selection of an input $i_f \in I_f$. It may be that the elements in I_f are inputs that are rarely used, and that therefore the software will be perceived as reliable.

Statistical testing may be used to make inferences on the future performance of the software. This requires an understanding of the expected usage profile of the system, as well as the population of all possible usages of the software. The sampling is done in accordance with the expected usage profile.

12.4.1   Software Reliability and Defects

The release of an unreliable software product may result in damage to property or injury (including loss of life) to a third party. Consequently, companies need to be confident that their software products are fit for use prior to their release.

Table 12.7 Adam's 1984 study of software failures of IBM products

	Rare				Frequent			
	1	2	3	4	5	6	7	8
MTTF (years)	5,000	1,580	500	158	50	15.8	5	1.58
Avg (%) fixes	33.4	28.2	18.7	10.6	5.2	2.5	1.0	0.4
Prob failure	0.008	0.021	0.044	0.079	0.123	0.187	0.237	0.300

This often involves setting objective product quality criteria to be satisfied prior to release. This provides a degree of confidence that the software has the desired quality, and is safe and fit for purpose. However, these results are historical in the sense that they are a statement of the past and the present quality. The problem is whether the past behaviour of the software provides a sound indication of its future behaviour.

Software reliability models are an attempt to predict the future reliability of the software, and to assist in deciding on whether the software is ready for release.

A defect does not always result in a failure, as it may be benign and may occur on a rarely used execution path. Many observed failures arise from a small proportion of the existing defects. Adam's 1984 case study [Ada:84] indicated that over 33 % of the defects led to an observed failure with mean time to failure greater than 5,000 years; whereas less than 2 % of defects led to an observed failure with a mean time to failure of less than 50 years. This suggests that a small proportion of defects led to almost all of the observed failures (Table 12.7).

The analysis shows that 61.6 % of all fixes (Group 1 and 2) were made were for failures that will be observed less than once in 1,580 years of expected use, and that these constitute only 2.9 % of the failures observed by typical users. On the other hand, groups 7 and 8 constitute 53.7 % of the failures observed by typical users and only 1.4 % of fixes.

The analysis showed that *coverage testing* is not cost effective in increasing MTTF. *Usage testing*, in contrast, would allocate 53.7 % of the test effort to fixes that will occur 53.7 % of the time for a typical user. Harlan Mills has argued [CoM:90] that the data in the table shows that usage testing is 21 times more effective than coverage testing

There is a need to be careful with *reliability growth models*, as there is no tangible growth in reliability unless the corrected defects are likely to manifest themselves as a failure.[1] Many existing software reliability growth models assume that all remaining defects in the software have an equal probability of failure, and that the correction of a defect leads to an increase in software reliability. These assumptions are questionable.

The defect count and defect density may be poor predictors of operational reliability. An emphasis on removing a large number of defects from the software may not be sufficient in itself to achieve high reliability.

---

[1] We are assuming that the defect has been corrected perfectly with no new defects introduced by the changes made.

Table 12.8 New and Old Version of Software

Relationship between New/Old Version of Software
The new version of the software is identical to the previous version except that the identified defects have been corrected.
The new version of the software is identical to the previous version, except that the identified defects have been corrected but the developers have introduced some new defects.
No assumptions can be made about the behaviour of the new version of the software until further data is obtained.

The correction of defects in the software leads to newer versions of the software, and reliability models assume reliability growth: i.e. the new version is more reliable than the older version as several identified defects have been corrected. However, in some sectors (such as the safety critical field), the view is that the new version of a program is a new entity, and that no inferences may be drawn until further investigation has been done. The relationship between the new version and the previous version of the software needs to be considered (Table 12.8).

The safety critical industry (e.g. the nuclear power industry) takes the conservative viewpoint that any change to a program creates a new program. The new program is therefore required to demonstrate its reliability.

12.4.2   Cleanroom Methodology

Harlan Mills and others at IBM developed the Cleanroom methodology to assist in the development of high-quality software. The software is released only when the probability of zero-defects is very high.

The way in which the software is used will impact upon its perceived quality and reliability. Failures will manifest themselves on certain input sequences only, and as users will generally employ different input sequences, each user will have a different perception of the reliability of the software. Knowledge of the way that the software will be used allows the software testing to be focused on verifying the correctness of the common everyday tasks carried out by the users.

This means that it is important to determine the operational profile of users to allow effective testing of the software to take place. The operational environment may not be stable as users may potentially change their behaviour over time. The collection of operational data involves identifying the operations to be performed, and the probability of that operation being performed.

The Cleanroom approach [CoM:90] applies statistical techniques to enable a software reliability measure to be calculated based upon the expected usage of the software. It employs *statistical usage testing* rather than coverage testing, and applies statistical quality control to certify the mean time to failure of the software. The statistical usage testing involves executing tests chosen from the population of all possible uses of the software in accordance with the probability of expected use.

Table 12.9 Characteristics
of Good Software Reliability
Model

Characteristics of Good Software Reliability Model
Good theoretical foundation
Realistic assumptions
Good empirical support
As simple as possible (Ockhams Razor)
Trustworthy and accurate.

Coverage testing involves designing tests that cover every path through the program, and this type of testing is as likely to find a rare execution failure as well as a frequent execution failure. It is highly desirable to find failures that occur in frequently used parts of the system.

The advantage of usage testing (that matches the actual execution profile of the software) is that it has a better chance of finding execution failures on frequently used parts of the system. This helps to maximise the expected mean time to failure.

12.4.3 Software Reliability Models

Models are simplifications of the reality and a good model allows accurate predictions of future behaviour to be made. The adequacy of the model is judged by model exploration, and determining if its predictions are close to the actual manifested behaviour. More accurate models are sought to replace inadequate models.

A model is judged effective if the empirical evidence supports it. Models are often modified (or replaced) over time, as further facts and observations lead to aberrations that cannot be explained by the current model. A good software reliability model will have the following characteristics (Table 12.9).

There are several software reliability predictor models currently employed (with varying degrees of success). Some of them just compute defect counts rather than estimating software reliability in terms of mean time to failure. They include:

- *Size and Complexity Metrics*
  These are used to predict the number of defects that a system will reveal in operation or testing.
- *Operational Usage Profile*
  These predict failure rates based on the expected operational usage profile of the system. The number of failures encountered is determined and the software reliability predicted.
- *Quality of the Development Process*
  These predict failure rates based on the process maturity of the software development process (e.g. CMMI level) in the organisation (Table 12.10).

The extent to which the software reliability model can be trusted depends on the accuracy of its predictions. Empirical data will need to be gathered to determine its accuracy. It may be acceptable to have a little inaccuracy during the early stages

Table 12.10 Software reliability models

Model	Description	Comments
Jelinski and Morandal	The failure rate is a Poisson process and is proportional to the current defect content of the program. The initial defect count is N; the initial failure rate is $N\varphi$; it decreases to $(N-1)\varphi$ after the first fault is detected and eliminated, and so on. The constant φ is termed the proportionality constant	Assumes that defects corrected perfectly and no new defects are introduced Assumes that each fault contributes the same amount to failure rate
Littlewood and Verrall	The successive execution time between failures are independent exponentially distributed random variables. Software failures are the result of the particular inputs and faults introduced from the correction of defects	Does not assume perfect correction of defects
Seeding and Tagging	This is analogous to estimating the fish population of a lake (Mills). A known number of defects is inserted into a software program and the proportion of these identified during testing is determined	Estimates the total number of defects in the software but not a software reliability predictor
	Another approach (Hyman) is to regard the defects found by one tester as tagged and then to determine the proportion of tagged defects found by a second independent tester	Assumes that all faults are equally likely to be found and introduced faults representative of existing
Generalised Poisson	The number of failures observed in ith time interval τ_i has a Poisson distribution with mean $\phi(N - M_{i-1}) \tau_i^{\alpha}$ where N is the initial number of faults, M_{i-1} is the total number of faults removed up to the end of the $(i-1)$th time interval; and ϕ is the proportionality constant	Assumes that the faults removed perfectly at end of time interval

of prediction, provided the predictions of operational reliability are close to the observations. A model that gives overly optimistic results is termed 'optimistic', whereas a model that gives overly pessimistic results is termed 'pessimistic'.

The assumptions in the reliability model need to be examined to determine whether they are realistic. Several software reliability models have questionable assumptions such as:

- All defects are corrected perfectly.
- Defects are independent of one another.
- Failure rate decreases as defects are corrected.
- Each fault contributes the same amount to the failure rate.

12.5   Review Questions

1. What is probability? What is statistics? Explain the difference between them.
2. Explain the laws of probability.
3. What is a sample space? What is an event?
4. Prove Boole's inequality $P(\cup_{i=1}^{n} E_i) \leq \sum_{i=1}^{n} P(E_i)$ where the E_i are not necessarily disjoint.
5. A couple has two children. What is the probability that both are girls if the eldest is a girl?
6. What is a random variable?
7. Explain the difference between the probability mass function and the probability density function (for both discrete and continuous random variables).
8. Explain variance, covariance and correlation.
9. Describe how statistics may be abused.
10. What is a random sample? Describe the methods available to generate a random sample from a population. How may information be gained from a sample?
11. Explain how the average of a sample may be determined, and discuss the mean, mode and median of a sample.
12. Explain sample variance and sample standard deviation.
13. Describe the normal distribution and the central limit theorem.
14. Describe hypothesis testing and acceptance or rejection of the null hypothesis.
15. What is software reliability? Describe various software reliability models.

12.6   Summary

Statistics is an empirical science that is concerned with the collection, organisation, analysis and interpretation and presentation of data. The data collection needs to be planned and this may include surveys and experiments. Statistics is widely used by government and industrial organisations. It may be used for forecasting as well as for presenting trends. Statistical sampling allows the behaviour of a random sample to be studied and inferences to be made about the population.

Probability theory provides a mathematical indication of the likelihood of an event occurring, and the probability is a numerical value between 0 and 1. A probability of 0 indicates that the event cannot occur, whereas a probability of 1 indicates that the event is guaranteed to occur. If the probability of an event is greater than 0.5, then this indicates that the event is more likely to occur than not.

   Software has become increasingly important for society and professional software companies aspire to develop high-quality and reliable software. Software Reliability is the probability that the program works without failure for a speci ed length of time, and is a statement on the future behaviour of the software. It is generally expressed in terms of the mean time to failure (MTTF) or the mean time between failure (MTBF), and the software reliability measurements are an attempt to provide an objective judgment of the tness for use of the software.

   There are many reliability models in the literature and the question as to which is the best model or how to evaluate the effectiveness of the model arises. A good model will have good theoretical foundations and will give useful predictions of the reliability of the software.

Chapter 13
Matrix Theory

Key Topics
Matrix
Matrix Operations
Inverse of a Matrix
Determinant
Eigenvectors and Eigenvalues
Cayley Hamilton Theorem
Cramer's Rule

13.1 Introduction

A *matrix* is a rectangular array of numbers that consists of horizontal rows and vertical columns. A matrix with m rows and n columns is termed an $m \times n$ matrix, where m and n are its dimensions. A matrix with an equal number of rows and columns (e.g. n rows and n columns) is termed a square matrix. The example matrix below is a square matrix with four rows and four columns.

The entry in the ith row and the jth column of a matrix A is denoted by A[i, j], $A_{i,j}$, or a_{ij}, and the matrix A may be denoted by the formula for its (i, j)th entry, i.e. (a_{ij}) where i ranges from 1 to m and j ranges from 1 to n (Fig. 13.1).

An $m \times 1$ matrix is termed a column vector, and a $1 \times n$ matrix is termed a row vector. Any row or column of a $m \times n$ matrix determines a row or column vector which is obtained by removing the other rows (respectively columns) from the matrix. For example, the row vector (11,−5, 5, 3) is obtained from the matrix example by removing rows 1, 2 and 4 of the matrix.

Two matrices A and B are equal if they are both of the same dimensions, and if $a_{ij} = b_{ij}$ for each $i = 1, 2, \ldots, m$ and each $j = 1, 2, \ldots, n$.

Matrices can be added and multiplied (provided certain conditions are satisfied). An addative identity matrix exists such that the addition of it to any matrix A yields A,

Fig. 13.1 Example of a
4 × 4 square matrix

$$\begin{pmatrix} 6 & 0 & -2 & 3 \\ 4 & 2 & 3 & 7 \\ 11 & -5 & -5 & 3 \\ 3 & -5 & -8 & 1 \end{pmatrix}$$

and similarly a multiplicative identity matrix I exists is such that $AI = IA = A$ for any matrix A. Square matrices have inverses (provided that their determinant is non-zero), and every matrix satisfies its characteristic polynomial.

It is possible to consider matrices with infinite rows and columns, and although it is not possible to write down such matrices explicitly it is still possible to add, subtract and multiply by a scalar provided there is a well-defined entry in each (i, j)th element of the matrix.

Matrices are an example of an algebraic structure known as an *algebra*. The discussion on coding theory in Chap. 9 introduced several algebraic structures such as groups, rings, fields and vector spaces. The matrix algebra for $m \times n$ matrices A, B and C and scalers λ and μ satisfies the following properties:

1. $A + B = B + A$ (Commutativity)
2. $A + (B + C) = (A + B) + C$ (Associativity)
3. $A + 0 = 0 + A = A$ (Additive Identity)
4. $A + (-A) = (-A) + A = 0$ (Additive Inverse)
5. $\lambda(A + B) = \lambda A + \lambda B$
6. $(\lambda + \mu)A = \lambda A + \mu B$
7. $\lambda(\mu A) = (\lambda\mu)A$
8. $1A = A$

Matrices have many applications including their use in graph theory to keep track of the distance between pairs of vertices in the graph. A rotation matrix may be employed to represent the rotation of a vector in three-dimensional space. The product of two matrices represents the composition of two linear transformations. Matrices may be employed to determine the solution to a set of linear equations. They also are used in computer graphics and may be employed to project a three-dimensional image onto a two-dimensional screen. It is essential to employ efficient algorithms for matrix computation, and this is an active area of research in the field of numerical analysis.

13.1.1   2 × 2 Matrices

Matrices arose in practice as a means of solving a set of linear equations. One of the earliest examples of their use is in a Chinese text dating from between 300 BC and 200 AD. The Chinese text showed how matrices could be employed to solve simultaneous equations. Consider the set of equations:

$$ax + by = r$$
$$cx + dy = s$$

The coefficients of the linear equations in x and y above may be represented by the matrix A, where A is given by:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

The linear equations may be represented as the multiplication of the matrix A and a vector x resulting in a vector v:

$$Ax = v$$

The matrix representation of the linear equations and its solution are as follows:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r \\ s \end{pmatrix}$$

The vector x may be calculated by determining the inverse of the matrix A (provided that its inverse exists). The vector x is then given by:

$$x = A^{-1}v$$

The solution to the set of linear equations is then given by:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} \begin{pmatrix} r \\ s \end{pmatrix}$$

The inverse of a matrix A exists if and only if its *determinant* is non-zero, and if this is the case, then the vector x is given by:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{\det A} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} r \\ s \end{pmatrix}$$

The determinant of a 2×2 matrix A is given by:

$$\det A = ad - cb.$$

The determinant of a 2×2 matrix is denoted by:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

A key property of determinants is that

$$\det(AB) = \det(A) \times \det(B)$$

The transpose of a 2×2 matrix A (denoted by A^T) involves exchanging rows and columns, and is given by:

$$A^T = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

The inverse of the matrix A (denoted by A^{-1}) is given by:

$$A^{-1} = \frac{1}{\det A} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Further, $AA^{-1} = A^{-1}A = I$ where I is the identity matrix of the algebra of 2×2 matrices under multiplication. That is:

$$AA^{-1} = A^{-1}A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The addition of two 2×2 matrices A and B is given by a matrix whose entries are the addition of the individual components of A and B. The addition of two matrices is commutative and we have:

$$A + B = A + B = \begin{pmatrix} a+p & b+q \\ c+r & d+s \end{pmatrix}$$

where A and B are given by:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad B = \begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

The identity matrix under addition is given by the matrix whose entries are all 0, and it has the property that $A + 0 = 0 + A = A$.

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

The multiplication of two 2×2 matrices is given by:

$$AB = \begin{pmatrix} ap+br & aq+bs \\ cp+dr & cq+ds \end{pmatrix}$$

The multiplication of matrices is not commutative, i.e. $AB \neq BA$. The multiplicative identity matrix I has the property that $AI = IA = A$ and it is given by:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

A matrix A may be multiplied by a scaler λ, and this yields the matrix λA where each entry in A is multiplied by the scaler λ. That is the entries in the matric λA are λa_{ij}.

$$\left(\begin{array}{ccccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{array}\right) \left(\begin{array}{cccc} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ b_{31} & b_{32} & \cdots & b_{3p} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{array}\right) = \left(\begin{array}{cccc} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ c_{31} & c_{32} & \cdots & c_{3p} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{array}\right)$$

<p align="center">m rows, n columns   n rows, p columns   m rows, p columns</p>

Fig. 13.2 Multiplication of two matrices

13.2 Matrix Operations

More general sets of linear equations may be solved with $m \times n$ matrices (i.e. a matrix with m rows and n columns) or square $n \times n$ matrices. In this section, we extend the discussion on 2×2 matrices to consider operations on more general matrices.

The addition and subtraction of two matrices A, B is meaningful if and only if A and B have the same dimensions, i.e. they are both $m \times n$ matrices. In this case, $A + B$ is defined by adding the corresponding entries:

$$(A + B)_{ij} = A_{ij} + B_{ij}$$

The additive identity matrix for the square $n \times n$ matrices is an $n \times n$ matrix whose entries are zero i.e. $r_{ij} = 0$ for all i, j where $1 \leq i \leq n$ and $1 \leq j \leq n$.

The scalar multiplication of a matrix A by a scalar k is meaningful and the resulting matrix kA is given by:

$$(kA)_{ij} = ka_{ij}$$

The multiplication of two matrices A and B is meaningful if and only if the number of columns of A is equal to the number of rows of B i.e. A is a $m \times n$ matrix and B is a $n \times p$ matrix and the resulting matrix AB is a $m \times p$ matrix (Fig. 13.2)

Let $A = (a_{ij})$ where i ranges from 1 to m and j ranges from 1 to n, and let $B = (b_{jl})$ where j ranges from 1 to n and l ranges from 1 to p. Then AB is given by (c_{il}) where i ranges from 1 to m and l ranges from 1 to p with c_{il} given by:

$$c_{il} = \sum_{k=1}^{n} a_{ik} b_{kl}.$$

That is, the entry (c_{il}) is given by multiplying the ith row in A by the lth column in B followed by a summation. Matrix multiplication is not commutative i.e. $AB \neq BA$.

Fig. 13.3 Identity matrix I_n

$$\begin{pmatrix} 1 & 0 & 0 & .... & 0 \\ 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & .... & 0 \\ ... & .... & .... & .... & .... \\ ... & .... & .... & .... & .... \\ ... & .... & .... & .... & .... \\ 0 & 0 & 0 & .... & 1 \end{pmatrix}$$

The multiplicative identity matrix I is a $n \times n$ matrix and the entries are given by r_{ij} where $r_{ii} = 1$ and $r_{ij} = 0$ where $i \neq j$. A matrix that has non-zero entries only on the diagonal is termed a *diagonal matrix*. A triangular matrix is a square matrix in which all the entries above or below the main diagonal are zero. A matrix is an *upper triangular* matrix if all entries below the main diagonal are zero, and *lower triangular* if all of the entries above the main diagonal are zero. Upper triangular and lower triangular matrices form a subalgebra of the algebra of square matrices (Fig. 13.3).

A key property of the identity matrix is that for all $n \times n$ matrices A we have:

$$AI = IA = A$$

The inverse of a $n \times n$ matrix A is a matrix A^{-1} such that:

$$AA^{-1} = A^{-1}A = I$$

The inverse A^{-1} exists if and only if the determinant of A is non-zero.

The *transpose* of a matrix $A = (a_{ij})$ involves changing the rows to columns and vice versa to form the transpose matrix A^T. The result of the operation is that the $m \times n$ matrix A is converted to the $n \times m$ matrix A^T. It is defined by (Fig. 13.4):

$$(A^T)_{ij} = (a_{ji}) \quad 1 \leq j \leq n \quad \text{and} \quad 1 \leq i \leq m$$

A matrix is *symmetric* if it is equal to its transpose i.e. $A = A^T$.

13.3   Determinants

The determinant is a function defined on square matrices and its value is a scalar. A key property of determinants is that a matrix is invertible if and only if its determinant is non-zero. The determinant of a 2×2 matrix is given by:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

$$
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\
a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn}
\end{pmatrix}^{T}
=
\begin{pmatrix}
a_{11} & a_{21} & a_{31} & \cdots & a_{m1} \\
a_{12} & a_{22} & a_{32} & \cdots & a_{m2} \\
a_{13} & a_{23} & a_{33} & \cdots & a_{m3} \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
a_{1n} & a_{2n} & a_{3n} & \cdots & a_{mn}
\end{pmatrix}
$$

m rows, n columns   *n rows, m columns*

Fig. 13.4 Transpose of a matrix

Fig. 13.5 Determining the (i, j) minor of A

$$
\begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\
a_{31} & a_{32} & \cdots & a_{3j} & \cdots & a_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{n1} & a_{n2} & \cdots & a_{mj} & \cdots & a_{nn}
\end{pmatrix}
=
$$

i,j minor of A

The determinant of a 3×3 matrix is given by:

$$
\begin{vmatrix}
a & b & c \\
d & e & f \\
g & h & i
\end{vmatrix}
= aei + bfg + cdh - afh - bdi - ceg
$$

Cofactors Let A be an $n \times n$ matrix. For $1 \le i, j \le n$, the (i, j) *minor* of A is defined to be the $(n-1) \times (n-1)$ matrix obtained by deleting the ith row and jth column of A (Fig. 13.5).

The shaded row is the ith row and the shaded column is the jth column. These are both deleted from A to form the (i, j) minor of A, and this is a $(n-1) \times (n-1)$ matrix.

The (i, j) *cofactor* of A is definedto be $(-1)^{i+j}$ times the determinant of the (i, j) minor. The (i, j) cofactor of A is denoted by $K_{ij}(A)$.

The cofactor matrix of A (denoted by Cof A) is formed in this way where the (i, j)th element in the cofactor matrix is the (i, j) cofactor of A.

Definition of Determinant The determinant of a matrix is defined as:

$$
\det A = \sum_{i=1}^{n} A_{ij} K_{ij}
$$

In other words the determinant of A is determined by taking any row of A and multiplying each element by the corresponding cofactor and adding the results. The determinant of the product of two matrices is the product of their determinants.

$$\det(AB) = \det A \times \det B$$

Definition The *adjugate* of A is the $n \times n$ matrix $Adj(A)$ whose (i, j) entry is the (j, i) cofactor $K_{ji}(A)$ of A. That is, the adjugate of A is the transpose of the cofactor matrix of A.

Inverse of A The inverse of A is determined from the determinant of A and the adjugate of A. That is,

$$A^{-1} = \frac{1}{\det A}\text{Adj A} = \frac{1}{\det A}(\text{Cof } A)^{\text{T}}$$

A matric is invertible if and only if its determinant is non-zero, i.e. A is invertible if and only if $\det(A) \neq 0$.

Cramer's Rule Cramer's rule is a theorem that expresses the solution to a system of linear equations with several unknowns using the determinant of a matrix. There is a unique solution if the determinant of the matrix is non-zero.

For a system of linear equations of the $Ax = v$ where x and v are n-dimensional column vectors, then if $\det A \neq 0$ then the unique solution for each x_i is

$$x_i = \frac{\det U_1}{\det A}$$

where U_i is the matrix obtained from A by replacing the ith column in A by the v-column.

Characteristic Equation For every $n \times n$ matrix A there is a polynomial equation of degree n satisfied by A. The *characteristic polynomial* of A is a polynomial in x of degree n. It is given by:

$$cA(x) = \det(xI - A).$$

Cayley–Hamilton Theorem Every matrix A satisfies its characteristic polynomial, i.e. $p(A) = 0$ where $p(x)$ is the characteristic polynomial of A.

13.4   Eigenvectors and Eigenvalues

A number λ is an eigenvalue of a $n \times n$ matrix A if there is a non-zero vector v such that the following equation holds:

$$Av = \lambda v$$

The vector v is termed an eigenvector and the equation is equivalent to:

$$(A - \lambda I)v = 0$$

This means that $(A - \lambda I)$ is a zero divisor and hence it is not an invertible matrix. Therefore,

$$\det(A - \lambda I) = 0$$

The polynomial function $p(\lambda) = \det(A - \lambda I)$ is called the characteristic polynomial of A, and it is of degree n. The characteristic equation is $p(\lambda) = 0$, and as the polynomial is of degree n, there are at most n roots of the characteristic equation, and so there are at the most n eigenvalues.

The *Cayley–Hamilton theorem* states that every matrix satisfies its characteristic equation, i.e. the application of the characteristic polynomial to the matrix A yields the zero matrix.

$$p(A) = 0$$

13.5 Gaussian Elimination

Gaussian elimination with backward substitution is an important method used in solving a set of linear equations. A matrix is used to respresent the set of linear equations, and Gaussian elimination reduces the matrix to a *triangular* or *reduced form*, which may then be solved by backward substitution.

This allows the set of n linear equations (E_1 to E_n) defined below to be solved by applying operations to the equations to reduce the matrix to triangular form. This reduced form is easier to solve and it provides exactly the same solution as the original set of equations. The set of equations is defined as:

$$E_1 : a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$
$$E_2 : a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$
$$\vdots \quad \vdots \quad \vdots \qquad \vdots \quad \vdots$$
$$E_n : a_{n_1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

Three operations are permitted on the equations and these operations transform the linear system into a reduced form. They are:

(a) Any equation may be multiplied by a non-zero constant.
(b) An equation E_i may be multiplied by a constant and added to another equation E_j, with the resulting equation replacing E_j
(c) Equations E_i and E_j may be transposed with E_j replacing E_i and vice versa.

This method for solving a set of linear equations is best illustrated by an example, and we consider an example taken from [BuF:89]. Then the solution to a set of linear equations with four unknowns may be determined as follows:

$$E_1 : \quad x_1 + \quad x_2 \qquad + 3x_4 = 4$$
$$E_2 : \quad 2x_1 + \quad x_2 - \quad x_3 + \quad x_4 = 1$$
$$E_3 : \quad 3x_1 - \quad x_2 - \quad x_3 + 2x_4 = -3$$
$$E_4 : -x_1 + 2x_2 + 3x_3 - \quad x_4 = 4$$

First, the unknown x_1 is eliminated from E_2, E_3, and E_4 and this is done by replacing E_2 with $E_2 - 2E_1$; replacing E_3 with $E_3 - 3E_1$; and replacing E_4 with $E_4 + E_1$. The resulting system is

$$E_1 : \quad x_1 + x_2 \qquad + \quad 3x_4 = 4$$
$$E_2 : \qquad - \quad x_2 - \quad x_3 - 5x_4 = -7$$
$$E_3 : \qquad -4x_2 - \quad x_3 - 7x_4 = -15$$
$$E_4 : \qquad 3x_2 + 3x_3 + 2x_4 = 8$$

The next step is then to eliminate x_2 from E_3 and E_4. This is done by replacing E_3 with $E_3 - 4E_2$ and replacing E_4 with $E_4 + 3E_2$. The resulting system is now in triangular form and the unknown variable may be solved easily by backward substitution.

$$E_1 : \quad x_1 + x_2 \qquad + \quad 3x_4 = 4$$
$$E_2 : \qquad -x_2 - x_3 - \quad 5x_4 = -7$$
$$E_3 : \qquad 3x_3 + 13x_4 = 13$$
$$E_4 : \qquad -13x_4 = -13$$

The usual approach to Gaussian elimination is to do it with an augmented matrix. That is, the set of equations is a $n \times n$ matrix and it is augmented by the column vector to form the augmented $n \times n + 1$ matrix. Gaussian elimination is then applied to the matrix to put it into triangular form, and it is then easy to solve the unknowns.

The other approach to solving a set of linear equation is to employ Cramer's rule, which was discussed in Sect. 13.4.

13.6   Review Questions

1. Show how 2×2 matrices may be added and multiplied.
2. What is the additive identity for 2×2 matrices and the multiplicative identity?
3. What is the determinant of a 2×2 matrix?
4. Show that a 2×2 matrix is invertible if its determinant is non-zero.
5. Describe the matrix algebra for general matrices.
6. What is Cramer's rule?
7. Show how Gaussian elimination may be used to solve a set of linear equations.

13.7   **Summary**

A matrix is a rectangular array of numbers that consists of horizontal rows and vertical columns. A matrix with m rows and n columns is termed an $m \times n$ matrix, and m and n are its dimensions. A matrix with an equal number of rows and columns (e.g., n rows and n columns) is termed a square matrix.

Matrices arose in practice as a means of solving a set of linear equations, and one of the earliest examples of their use is from a Chinese text dating from between 300 BC and 200 AD.

Matrices of the same dimensions may be added, subtracted, and multiplied by a scalar. Two matrices A and B may be multiplied provided that the number of columns of A equals the number of rows in B).

A square matrix has an inverse provided that its determinant is non-zero. The inverse of a matrix involves determining its determinant, constructing the cofactor matrix, and transposing the cofactor matrix.

The solution to a set of linear equations may be determined by Gaussian elimination to convert the matrix to upper triangular form, and then employing backward substitution. Another approach is to use the Cramer's rule.

Eigenvalues and eigenvectors lead to the characteristic polynomial and every matrix satisfies its characteristic polynomial.

Chapter 14
Complex Numbers and Quaternions

Key Topics
 Complex Numbers
 Argand Diagram
 Polar Representation
 De Moivres Theorem
 Complex Conjugate
 Quaternions

14.1 Introduction

A complex number z is a number of the form $a + bi$ where a and b are real numbers and $i^2 = -1$. Cardona, who was a sixteenth century Italian mathematician, introduced complex numbers, and he used them to solve cubic equations. The set of complex numbers is denoted by $\mathbb{C}$, and each complex number has two parts namely the real part $\text{Re}(z) = a$, and the imaginary part $\text{Im}(z) = b$. The set of complex numbers is a superset of the set of real numbers, and this is clear since every real number is a complex number with an imaginary part of zero. A complex number with a real part of zero (i.e. $a = 0$) is termed an imaginary number. Complex numbers have many applications in physics, engineering and applied mathematics (Fig. 14.1).

A complex number may then be viewed as a point in a 2-dimensional Cartesian coordinate system (called the complex plane or Argand diagram), where the complex number $a + bi$ is represented by the point (a,b) on the complex plane. The real part of the complex number is the horizontal component, and the imaginary part is the vertical component.

Quaternions are an extension of complex numbers. A quaternion number is a quadruple of the form $(a + bi + cj + dk)$ where $i^2 = j^2 = k^2 = ijk = -1$. The set of quaternions is denoted by $\mathbb{H}$, and the quaternions form an algebraic system known as a division ring. The multiplication of two quaternions is not commutative, i.e. given q_1, $q_2 \in \mathbb{H}$ then $q_1, q_2 \neq q_2, q_1$. Quaternions were one of the first non-commutative algebraic structures to be discovered.

Fig. 14.1 Argand diagram
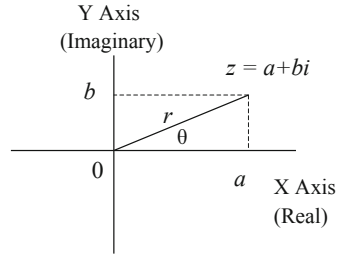
The Irish mathematician, Sir William Rowan Hamilton,[1] discovered *quaternions*. Hamilton was trying to generalise complex numbers to triples without success. He had a moment of inspiration along the banks of the Royal Canal in Dublin, and he realised that if he used quadruples instead of triples that a generalisation from the complex numbers to quadruples was possible. He was so overcome with emotion at his discovery that he traced the famous quaternion formula[2] on Brooms Bridge in Dublin. This formula is given by:

$$i^2 = j^2 = k^2 = ijk = -1$$

Quarternions have many applications in physics and quantum mechanics and are applicable to the computing field. They are useful and efficient in describing rotations and are therefore applicable to computer graphics, computer vision and robotics.

14.2   Complex Numbers

There are several operations on complex numbers such as addition, subtraction, multiplication, division, and so on. Consider two complex numbers $z_1 = a + bi$ and $z_2 = c + di$. The various operations are shown in Table 14.1.

Properties of Complex Numbers The absolute value of a complex number z is denoted by $|z| = \sqrt{(a^2 + b^2)}$, and is just its distance from the origin. It has the following properties:

  (i) $|z| \geq 0$ and $|z| = 0$ if and only if $z = 0$.
 (ii) $|z| = |z^*|$
(iii) $|z_1 + z_2| \leq |z_1| + |z_2|$ (This is known as the triangle inequality.)
 (iv) $|z_1 z_2| = |z_1| \, |z_2|$
  (v) $|1/z| = 1/|z|$
 (vi) $|z_1/z_2| = |z_1|/|z_2|$

---

[1] There is a possibility that the German mathematician, Gauss, discovered *quaternions* earlier.

[2] Eamonn DeValera, a former taoiseach and president of Ireland, was formerly a mathematics teacher, and his interests included maths physics and quaternions. He is alleged to have carved the quaternion formula on the door of his cell when in prison during the Irish struggle for independence from Britain.

Table 14.1 Operations on complex numbers

Operation	Definition
Addition	$z_1 + z_2 = (a + bi) + (c + di) = (a + c) + (b + d)i$. The addition of two complex numbers may be interpreted as the addition of two vectors.
Subtraction	$z_1 - z_2 = (a + bi) - (c + di) = (a - c) + (b - d)i$.
Multiplication	$z_1 \, z_2 = (a + bi) \cdot (c + di) = (ac - bd) + (ad + cb)\, i$
Division	This operation is defined for $z_2 \neq 0$ $\dfrac{z_1}{z_2} = \dfrac{a + bi}{c + di} = \dfrac{ac + bd}{c^2 + d^2} + \dfrac{bc - ad}{c^2 + d^2} i$
Conjugate	The conjugate of a complex number $z = a + bi$ is given by $z^* = a - bi$. Clearly, $z^{**} = z$ and $(z_1 + z_2)^* = z_1^* + z_2^*$ Further, $\mathrm{Re}(z) = z + z^*/2$ and $\mathrm{Im}(z) = z - z^*/2i$
Absolute value	The absolute value or modulus of a complex number $z = a + bi$ is given by $\|z\| = \sqrt{(a^2 + b^2)}$. Clearly, $z.z^* = \|z\|^2$
Reciprocal	The reciprocal of a complex number z is defined for $z \neq 0$ and is given by: $\dfrac{1}{z} = \dfrac{1}{a + bi} = \dfrac{a - bi}{a^2 + b^2} = \dfrac{z^*}{\|z\|^2}$

Proof (iii)

$$
\begin{aligned}
|z_1 + z_2|^2 &= (z_1 + z_2)(z_1 + z_2)^* \\
&= (z_1 + z_2)(z_1{}^* + z_2{}^*) \\
&= z_1 z_1{}^* + z_1 z_2{}^* + z_2 z_1{}^* + z_2 z_2{}^* \\
&= |z_1|^2 + z_1 z_2{}^* + z_2 z_1{}^* + |z_2|^2 \\
&= |z_1|^2 + z_1 z_2{}^* + (z_1 z_2{}^*)^* + |z_2|^2 \\
&= |z_1|^2 + 2\,\mathrm{Re}\,(z_1 z_2{}^*) + |z_2|^2 \\
&\leq |z_1|^2 + 2|z_1 z_2{}^*| + |z_2|^2 \\
&= |z_1|^2 + 2|z_1||z_2{}^*| + |z_2|^2 \\
&= |z_1|^2 + 2|z_1||z_2| + |z_2|^2 \\
&= (|z_1| + |z_2|)^2
\end{aligned}
$$

Therefore, $|z_1 + z_2| \leq |z_1| + |z_2|$ and so the triangle inequality is proved.

The modulus of z is used to define a distance function between two complex numbers, and $d(z_1, z_2) = |z_1 - z_2|$. This turns the complex numbers into a metric space.[3]

Interpretation of Complex Conjugate The complex conjugate of the complex number $z = a + bi$ is defined as $z^* = a - bi$, and this the reflection of z about the real axis is shown in Fig. 14.2.

---

[3] A non-empty set X with a distance function d is a metric space if(i) $d(x, y) \geq 0$ and $d(x, y) = 0$ Û $x = y$(ii) $d(z, y) = d(y, x)$(iii) $d(x, y) \leq d(x, z) + d(z, y)$

Fig. 14.2 Interpretation of
complex conjugate
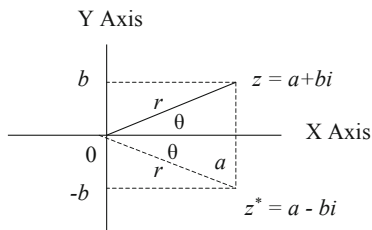
Fig. 14.3 Interpretation of
Eulers' formula
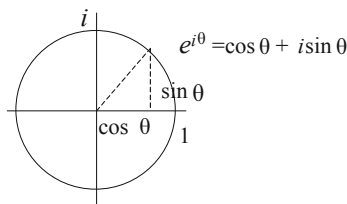
The modulus |z| of the complex number z is the distance of the point z from the
origin.

Polar Representation of Complex Numbers The complex number $z = a + bi$
may also be represented in polar form (r,θ) in terms of its modulus |z| and the
argument θ.

$$\cos\theta = \frac{a}{\sqrt{a^2 + b^2}} = \frac{a}{|z|}$$

$$\sin\theta = \frac{b}{\sqrt{a^2 + b^2}} = \frac{b}{|z|}$$

Let r denote the modulus of z: i.e., $r = |z|$. Then, z may be represented by
$(r\cos\theta + ir\sin\theta) = r(\cos\theta + i\sin\theta)$. Clearly, $\text{Re}(z) = r\cos\theta$ and $\text{Im}(z) = r\sin\theta$. Euler's formula (discussed below) states that $re^{i\theta} = r(\cos\theta + i\sin\theta)$.

The *principle argument* θ (denoted by Arg θ) is chosen so that $\theta \in [-\pi, \pi]$. There
is, of course, more than one argument θ that will satisfy $z = r(\cos\theta + i\sin\theta)$. In
fact, the full set of arguments (denoted by arg z) is given by arg $z = \theta + 2k\pi$, where
$k \in \mathbb{Z}$ and satisfies $z = re^{i(\theta + 2k\pi)}$.

Euler's Formula Euler's remarkable formula expresses the relationship between
the exponential function for complex numbers and trigonometric functions. It is
named after the eighteenth century Swiss mathematician, Euler.

The formula may be interpreted as the function $e^{i\theta}$ traces out the unit circle in
the complex plane as the angle θ ranges through the real numbers. Euler's formula
provides a way to convert between Cartesian coordinates and polar coordinates (r,θ)
(Fig. 14.3). It states that:

$$e^{i\theta} = \cos\theta + i\sin\theta$$

Further, the complex number $z = a + bi$ may be represented in polar coordinates as $z = r (\cos \theta + i \sin \theta) = re^{i\theta}$. Further,

$$e^z = e^a (\cos b + i \sin b) \quad \text{where } z = a + bi$$

Next, we prove Euler's formula: i.e., $e^{i\theta} = \cos \theta + i \sin \theta$.

Proof Recall the exponential expansion for e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^r}{r!} + \cdots$$

The expansion of $e^{i\theta}$ is then given by:

$$e^{i\theta} = 1 + i\theta + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \cdots + \frac{(i\theta)^r}{r!} + \cdots$$

$$= 1 + i\theta - \frac{\theta^2}{2!} - \frac{i\theta^3}{3!} + \frac{\theta^4}{4!} + \frac{i\theta^5}{5!} + \cdots + \frac{(i\theta)^r}{r!} + \cdots$$

$$= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \cdots\right) + i\left(\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \cdots\right)$$

$$= \cos \theta + i \sin \theta$$

(This follows from the Taylor series expansion of $\sin(\theta)$e and $\cos(\theta)$).

De Moivre's Theorem

$$(\cos \theta + i \sin \theta)^n = (\cos n\theta + i \sin n\theta) \quad \text{(where } n \in \mathbb{Z})$$

Proof This result is proved by mathematical induction and the result is clearly true for the base case $n = 1$.

Inductive Step: The inductive step is to assume that the theorem is true for $n = k$ and to then show that it is true for $n = k + 1$. That is, we assume that

$$(\cos \theta + i \sin \theta)^k = (\cos k\theta + i \sin k\theta) \quad \text{(for some } k > 1)$$

We next show that the result is true for $n = k + 1$:

$$(\cos \theta + i \sin \theta)^{k+1} = (\cos \theta + i \sin \theta)^k (\cos \theta + i \sin \theta)$$

$$= (\cos k\theta + i \sin k\theta)(\cos \theta + i \sin \theta) \quad \text{(from inductive step)}$$

$$= (\cos k\theta \cos \theta - \sin k\theta \sin \theta) + i(\cos k\theta \sin \theta + \sin k\theta \cos \theta)$$

$$= \cos (k\theta + \theta) + i \sin (k\theta + \theta)$$

$$= \cos (k + 1) \theta + i \sin (k + 1)\theta$$

Therefore, we have shown that if the result is true for some value of n say $n = k$, then the result is true for $n = k + 1$. We have shown that the base case of $n = 1$ is true, and it therefore follows that the result is true for $n = 2, 3, \ldots$ and for all natural numbers. The result may also be shown to be true for the integers.

Complex Roots   Suppose that z is a non-zero complex number and that n is a positive integer. Then z has exactly n distinct n^{th} roots that are given in polar form by:

$$\sqrt[n]{|z|}\left(\cos\left\{\frac{\text{Arg } z + 2k\pi}{n}\right\} + i\sin\left\{\frac{\text{Arg } z + 2k\pi}{n}\right\}\right)$$

for $k = 0, 1, 2, \ldots, n-1$

Proof   The objective is to find all complex numbers w such that $w^n = z$ where $w = |w|$ $(\cos\Phi + i\sin\Phi)$. Using De Moivre's Theorem this results in:

$$|w^n|(\cos n\Phi + i\sin n\Phi) = |z|(\cos\theta + i\sin\theta)$$

Therefore, $|w| = \sqrt[n]{|z|}$ and $n\Phi = \theta + 2k\pi$ for some k. That is,

$$\Phi = \frac{(\theta + 2k\pi)}{n} = \frac{(\text{Arg}z + 2k\pi)}{n}$$

The choices $k = 0, 1, \ldots, n-1$ produce the distinct nth roots of z.

Fundamental Theorem of Algebra   Every polynomial equation with complex coefficients has complex solutions, and the roots of a complex polynomial of degree n exist, and the n roots are all complex numbers.

Complex Derivatives   A function $f: A \rightarrow \mathbb{C}$ is said to be differentiable at a point z_0 if f is continuous at z_0 and if the limit below exists. The derivative at z_0 is denoted by $f'(z_0)$.

$$f'(z_0) = \lim_{z \to z_0} \frac{f(z) - f(z_0)}{z - z_0}$$

It is often written as

$$f'(z_0) = \lim_{h \to 0} \frac{f(z_0 + h) - f(z_0)}{h}$$

14.3   Quaternions

The Irish mathematician, Sir William Rowan Hamilton, discovered quaternions in the nineteenth century. Hamilton was born in Dublin in 1805 and died in 1865. He attended Trinity College, Dublin and was appointed professor of astronomy in 1827 while still an undergraduate. He made important contributions to optics, classical mechanics and mathematics (Fig. 14.4).

He discovered quaternions in 1843 while he was walking with his wife from his home at Dunsink Observatory to the Royal Irish Academy in Dublin. This route followed the towpath of the Royal Canal, and Hamilton had a sudden flash of inspiration on the idea of quaternion algebra at Broom's Bridge. He was so overcome with

Fig. 14.4 William Rowan
Hamilton

Fig. 14.5 Plaque at Broom's
Bridge

emotion at his discovery that he carved the quaternion formula into the stone on the
bridge. Today, there is a plaque at Broom's Bridge that commemorates Hamilton's
discovery (Fig. 14.5).

$$i^2 = j^2 = k^2 = ijk = -1$$

Quaternions are an extension of complex numbers. Hamilton had been trying for
many years to extend complex numbers to 3-dimensional space without success.
Complex numbers are numbers of the form $(a + bi)$ where $i^2 = -1$, and may be
regarded as points on a two-dimensional plane. A quaternion number is of the form
$(a + bi + cj + dk)$ where $i^2 = j^2 = k^2 = ijk = -1$, and may be regarded as points in
four-dimensional space.

The set of quaternions is denoted by $\mathbb{H}$. The quaternions form an algebraic system
known as a division ring. The multiplication of two quaternions is not commuta-
tive, i.e. given $q_1, q_2 \in \mathbb{H}$ then $q_1, q_2 \neq q_2, q_1$. Quaternions were one of the first
non-commutative algebraic structures to be discovered and other non-commutative
algebras (e.g. matrix algebra) were discovered in later years.

Quaternions have many applications in physics, quantum mechanics and theoret-
ical and applied mathematics. Gibbs and Heaviside later developed vector analysis,
and it replaced quaternions from the 1880s. Quaternions have become important in
computing in recent years, as they are useful and efficient in describing rotations.
They are applicable to computer graphics, computer vision and robotics.

14.3.1 Quaternion Algebra

Hamilton had been trying to extend the two-dimensional space of the complex numbers to a 3-dimensional space of triples. He wanted to be able to add, multiply and divide triples of numbers but he was unable to make progress on the problem of the division of two triples.

The generalisation of complex numbers to the 4-dimensions quaternions rather than triples allows the division of two quaternions to take place. A quarternion is of the form $(a + bi + cj + dk)$ where $1,i,j,k$ are the basis elements, and where the following properties are satisfied:

$$i^2 = j^2 = k^2 = ijk = -1 \quad \text{(Quaternion Formula)}$$

This formula leads to the following properties:

$$ij = k = -ji$$
$$jk = i = -kj$$
$$ki = j = -ik$$

These properties can be easily derived from the quaternion formula. For example:

$$ijk = -1$$
$$\Rightarrow ijkk = -k \quad \text{(Right multiplaying by } k)$$
$$\Rightarrow ij(-1) = -k \quad \text{(since } k^2 = -1)$$
$$\Rightarrow ij = k$$

Similarly, from

$$ij = k$$
$$\Rightarrow i^2 j = ik \quad \text{(Left multiplying by } i)$$
$$\Rightarrow -j = ik \quad \text{(since } i^2 = -1)$$
$$\Rightarrow j = -ik$$

Table 14.2 below represents the properties of quaternions under multiplication:

The quaternions ($\mathbb{H}$) form a division ring and quaternion multiplication is not commutative.

Addition and subtraction of Quaternions The addition of two quaternions $q_1 = (a_1 + b_1 i + c_1 j + d_1 k)$ and $q_2 = (a_2 + b_2 i + c_2 j + d_2 k)$ is given by

$$q_1 + q_2 = (a_1 + a_2) + (b_1 + b_2)i + (c_1 + c_2)j + (d_1 + d_2)k$$
$$q_1 - q_2 = (a_1 - a_2) + (b_1 - b_2)i + (c_1 - c_2)j + (d_1 - d_2)k$$

Table 14.2 Basic quaternion multiplication

$\times$	1	i	j	k
1	1	i	j	k
i	i	-1	k	$-j$
j	j	$-k$	-1	i
k	k	j	$-i$	-1

Identity Element   The addition identity is given by the quaternion $(0 + 0i + 0j + 0k)$ and the multiplicative identity is given by $(1 + 0i + 0j + 0k)$.

Multiplication of Quaternions   The multiplication of two quaternions q_1 and q_2 is determined by the product of the basis elements and the distributive law. It yields:

$$q_1 \cdot q_2 = a_1a_2 + a_1b_2i + a_1c_2j + a_1d_2k$$
$$+ b_1a_2i + b_1b_2ii + b_1c_2ij + b_1d_2ik$$
$$+ c_1a_2j + c_1b_2ji + c_1c_2jj + c_1d_2jk$$
$$+ d_1a_2k + d_1b_2ki + d_1c_2kj + d_1d_2kk$$

This may then be simplified to:

$$q_1 \cdot q_2 = a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2$$
$$+ (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i$$
$$+ (a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j$$
$$+ (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k$$

The multiplication of two quaternions may be defined in terms of matrix multiplication. It is easy to see that the product of the two quaternions above is equivalent to:

$$q_1q_2 = \begin{pmatrix} a_1 & -b_1 & -c_1 & -d_1 \\ b_1 & a_1 & -d_1 & c_1 \\ c_1 & d_1 & a_1 & -b_1 \\ d_1 & -c_1 & b_1 & a_1 \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \end{pmatrix}$$

This may also be written as:

$$(a_2\ b_2\ c_2\ d_2) \begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ -b_1 & a_1 & d_1 & -c_1 \\ -c_1 & -d_1 & a_1 & b_1 \\ -d_1 & c_1 & -b_1 & a_1 \end{pmatrix} = q_1q_2$$

Property of Quaternions under Multiplication   The quaternions are *not commutative* under multiplication. That is,

$$q_1q_2 \neq q_2q_1$$

The quaternions are associative under multiplication. That is,

$$q_1(q_2q_3) = (q_1q_2)q_3$$

Conjugation The conjugation of a quaternion is analogous to the conjugation of a complex number. The conjugate of a complex number $z = (a + bi)$ is given by $z^* = (a - bi)$. Similarly, the conjugate of a quaternion is determined by reversing the sign of the vector part of the quaternion. That is, the conjugate of $q = (a + bi + cj + dk)$ (denoted by q^*) is given by $q^* = (a - bi - cj - dk)$.

Scalar and Vector Parts A quaternion $(a + bi + cj + dk)$ consists of a *scalar* part a and a *vector* part $bi + cj + dk$. The scalar part is always real and the vector part is imaginary. That is, the quaternion q may be represented $q = (s,v)$ where s is the scalar part and v is the vector part. The scalar part of a quaternion is given by $q + q^*/2$ and the vector part is given by $q - q^*/2$. The *norm* of a quaternion q (denoted by $\|q\|$) is given by:

$$\|q\| = \sqrt{qq^*} = \sqrt{q^*q} = \sqrt{a^2 + b^2 + c^2 + d^2}$$

A quaternion of norm one is termed a unit quaternion (i.e., $\|u\| = 1$). Any quaternion u is defined by $u = q/\|q\|$ is a unit quaternion. Given $\alpha \in \mathbb{R}$ then $\|\alpha q\| = |\alpha|\,\|q\|$. The inverse of a quaternion q is given by q^{-1} where

$$q^{-1} = \frac{q^*}{\|q\|^2}$$

and $qq^{-1} = q^{-1}q = 1$

Given two quaternions p and q we have:

$$\|pq\| = \|p\|\,\|q\|$$

The norm is used to define the distance between two quaternions p and q (denoted by $d(p,q)$ and is given by:

$$d(p,q) = \|p - q\|$$

Representing Quaternions with 2×2 Matrices over Complex Numbers The quaternions have an interpretation under the 2×2 matrices where the basis elements i,j,k may be interpreted as matrics. Recall that the multiplicative identity for 2×2 matrices is

$$1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad -1 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$

Consider then the quaternion basis elements defined as follows:

$$i = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad j = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} \quad k = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}$$

Then a simple calculation shows that:

$$i^2 = j^2 = k^2 = ijk = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} = -1$$

Then the quaternion $q = (a + bi + cj + dk)$ may also be defined as

$$a \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + b \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} + c \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} + d \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}$$

This may be simplified to the complex matrix

$$q = \begin{pmatrix} a + di & b + ci \\ -b + ci & a - di \end{pmatrix}$$

and this is equivalent to:

$$q = \begin{pmatrix} u & v \\ -v^* & u^* \end{pmatrix}$$

where $u = a + di$ and $v = b + ci$.

The addition and multiplication of quaternions is then just the usual matrix addition and multiplication. Quaternions may also be represented by 4×4 real matrices.

14.3.2   *Quaternions and Rotations*

Quaternions may be applied to computer graphics, computer vision and robotics, and unit quaternions provide an efficient mathematical way to represent rotations in 3-dimensions. They offer an alternative to Euler angles and matrices.

The unit quaternion $q = (s, v)$ that computes the rotation about the unit vector u by an angle θ is given by:

$$(\cos(\theta/2), u \, \sin(\theta/2))$$

The scalar part is given by $s = \cos(\theta/2)$ and the vector part is given by $v = u \sin(\theta/2)$.

A point p in space is represented by the quaternion $P = (0, p)$. The result of the rotation of p is given by:

$$P_q = q P q^{-1}$$

Suppose we have two rotations represented by the unit quaternions q_1 and q_2, and we first perform q_1 followed by the rotation q_2. Then the composition of the two rotations is given by applying q_2 to the result of applying q_1. This is given by the following:

$$\begin{aligned} P(q_2 \text{ o } q_1) &= q_2(q_1 P q_1^{-1})q_2^{-1} \\ &= q_2 q_1 P q_1^{-1} q_2^{-1} \\ &= (q_2 q_1) P (q_2 q_1)^{-1} \end{aligned}$$

14.4   Review Questions

1. What is a complex number?
2. Show how a complex number may be represented as a point on the complex plane.
3. Show that $|z_1 z_2| = |z_1||z_2|$
4. Evaluate the following
   (a) $(-1)^{1/4}$
   (b) $1^{1/5}$
5. What is the fundamental theorem of algebra?
6. Show that $d/dz\ z^n = nz^{n-1}$
7. What is a quaternion?
8. Investigate the application of quaternions to computer graphics and robotics.

14.5   Summary

A complex number z is a number of the form $a + bi$ where a and b are real numbers and $i^2 = -1$. The set of complex numbers is denoted by $\mathbb{C}$, and a complex number has two parts, namely its real part a and imaginary part b. The complex numbers are an extension of the set of real numbers, and those with a real part $a = 0$ are termed imaginary numbers. Complex numbers have many applications in physics, engineering and applied mathematics.

Sir William Rowan Hamilton discovered the quaternions. He had spent many years trying to extend the two-dimensional space of the complex numbers to a 3-dimensional space of triples without success. He wanted to be able to add, multiply and divide triples of numbers, but he was unable to make progress on the problem of the division of two triples. His insight was that if he considered quadruples rather than triples then this structure would give him the desired mathematical properties. Hamilton also made important contributions to optics, classical mechanics and mathematics.

The generalisation of complex numbers to the 4-dimensions quaternions rather than triples allows the division of two quaternions. The quarternion is a number of the form $(a + bi + cj + dk)$ where $1,i,j,k$ are the basis elements (where 1 is the identity) and satisfy the quaternion formula:

$$i^2 = j^2 = k^2 = ijk = -1$$

Quarternions have many applications in physics and quantum mechanics. Quaternions have become important in computing in recent years as they are useful and efficient in describing rotations. They are applicable to computer graphics, computer vision and robotics.

Chapter 15
Calculus

Key Topics
Limit of a Function
Mean Value Theorem
Taylor's Theorem
Differentiation
Maxima and Minima
Integration
Numerical Analysis
Fourier Series
Laplace Transforms
Differential Equations

15.1 Introduction

Newton and Leibniz independently developed calculus in the late seventeenth century.[1] It plays a key role in describing how rapidly things change and may be employed to calculate the areas of regions under curves, the volumes of figures, and in finding tangents to curves. It is an important branch of mathematics concerned with limits, continuity, derivatives and integrals of functions.

The concept of a *limit* is fundamental in calculus. Let f be a function defined on the set of real numbers, then the limit of f at a is l (written as $\lim_{x \to a} f(x) = l$) if given any real number $\varepsilon > 0$ then there exists a real number $\delta > 0$ such that $|f(x) - l| < \varepsilon$ whenever $|x - a| < \delta$. The idea of a limit can be seen in Fig. 15.1.

---

[1] The question of who first invented calculus led to a bitter controversy between Newton and Leibniz, with the latter accused of plagiarising Newton's work. Newton an English mathematician and physicist was the giant of the late seventeenth century, and Leibnitz was a German mathematician and philosopher. Today, both Newton and Leibniz are credited with the independent development of calculus.

Fig. 15.1 Limit of a function
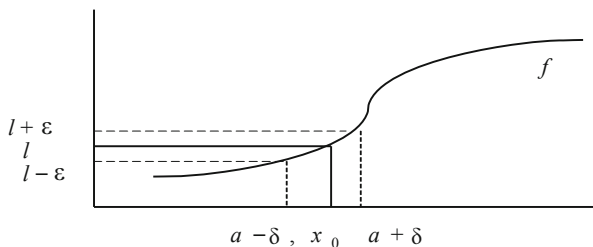
Fig. 15.2 Derivative as a
tangent to curve
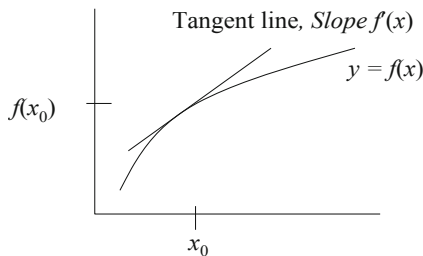
The function f defined on the real numbers is *continuous* at a if $\lim_{x \to a} f(x) = f(a)$. The set of all continuous functions on the closed interval (a, b) is denoted by $C(a, b)$.

If f is a function defined on an open interval containing x_0 then f is said to be *differentiable* at x_0 if the limit

$$\lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

exists. Whenever this limit exists it is denoted by $f'(x_0)$ and is called the *derivative* of f at x_0. Differential calculus is concerned with the properties of the derivative of a function. The derivative of f at x_0 is the slope of the tangent line to the graph of f at $(x_0, f(x_0))$ (Fig. 15.2).

It is easy to see that if a function f is differentiable at x_0 then f is continuous at x_0.

Theorem 15.1 (Rolle's Theorem) Suppose $f \in C(a, b)$ and f is differentiable on (a, b). If $f(a) = f(b)$ then there exists c such that $a < c < b$ with $f'(c) = 0$.

Theorem 15.2 (Mean Value Theorem) Suppose $f \in C(a, b)$ and f is differentiable on (a, b). Then there exists c such that $a < c < b$ with

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

Proof The mean value theorem is a special case of Rolle's theorem and the proof involves defining the function $g(x) = f(x) - rx$ where $r = (f(b) - f(a))/(b - a)$.

It is easy to verify that $g(a) = g(b)$. Clearly, g is differentiable on (a, b) and so by Rolle's theorem there is a c in (a, b) such that $g'(c) = 0$. Therefore, $f'(c) - r = 0$ and so $f'(c) = r = f(b) - f(a)/(b - a)$. as required.

Fig. 15.3 Interpretation of
Mean Value Theorem
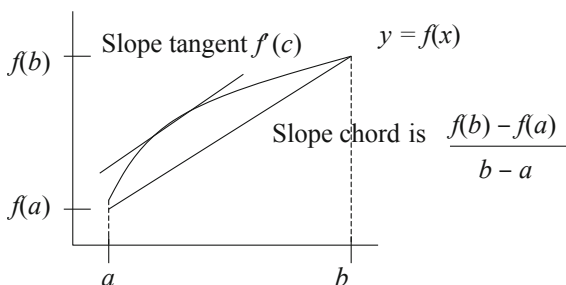
Fig. 15.4 Interpretation of
Intermediate Value Theorem
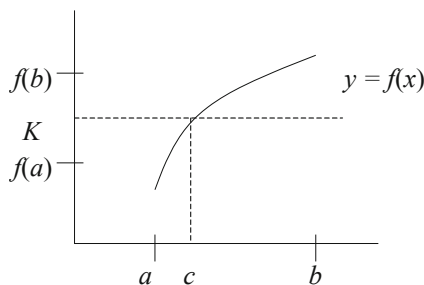
Interpretation of the Mean Value Theorem The mean value theorem essentially states that there is at least one point c between a and b, such that the slope of the chord between $(a, f(a))$ and $(b, f(b))$ is the same as the tangent at $(c, f'(c))$ (Fig. 15.3).

Theorem 15.3 (Intermediate Value Theorem) Suppose $f \in C(a, b)$ and K is any real number between $f(a)$ and $f(b)$. Then there exists c in (a, b) for which $f(c) = K$ (Fig. 15.4).

Proof The proof of this relies on the completeness property of the real numbers. It involves considering the set S in (a, b) such that $f(x) \leq K$ and noting that this set is non-empty since $a \in S$ and bounded above by b. Therefore, the supremum[2] $\sup S = c$ exists and it is straightforward to show (using ε and δ arguments and the fact that f is continuous) that $f(c) = K$.

L'Hôpital's Rule Suppose that $f(a) = g(a) = 0$ and that $f'(a)$ and $g'(a)$ exist and that $g'(a) \neq 0$. Then L'Hopital's Rule states that:

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \frac{f'(a)}{g'(a)}$$

Proof

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \lim_{x \to a} \frac{f(x) - f(a)}{g(x) - g(a)}$$

---

[2] The supremum is the least upper bound and the infinum is the greatest lower bound.

$$= \lim_{x \to a} \frac{\dfrac{f(x) - f(a)}{x - a}}{\dfrac{g(x) - g(a)}{x - a}}$$

$$= \frac{\displaystyle\lim_{x \to a} \frac{f(x) - f(a)}{x - a}}{\displaystyle\lim_{x \to a} \frac{g(x) - g(a)}{x - a}}$$

$$= \frac{f'(a)}{g'(a)}$$

Theorem 15.4 (Taylor's Theorem)   The Taylor series is concerned with the approximation to values of the function f near x_0. The approximation employs a polynomial (or power series) in powers of $(x - x_0)$ as well as the derivatives of f at $x = x_0$. There is an error term (or remainder) associated with the approximation.

Proof Suppose $f \in C^n(a, b)$ and f^{n+1} exists on (a, b). Let $x_0 \in (a, b)$ then for every $x \in (a, b)$ there exists $\zeta(x)$ between x_0 and x with

$$f(x) = P_n(x) + R_n(x)$$

where $P_n(x)$ is the n-th Taylor polynomial for f about x_0 and $R_n(x)$ is the called the remainder term associated with $P_n(x)$. The infinite series obtained by taking the limit of $P_n(x)$ as $n \to \infty$ is termed the Taylor series for f about x_0.

$$P_n(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^n(x_0)}{n!}(x - x_0)^n$$

The remainder term is give by:

$$R_n(x) = \frac{f^{n+1}(\xi(x))}{(n + 1)!}(x - x_0)^{n+1}$$

15.2   Differentiation

Mathematicians of the seventeenth century were working on various problems concerned with motion. These included problems such as determining the motion or velocity of objects on or near the earth, as well as the motion of the planets. They were also interested in changes of motion, i.e. in the acceleration of these moving bodies (Fig. 15.5).

Speed is the rate at which distance changes with respect to time, and the average speed during a journey is the distance travelled divided by the elapsed time. However, since the speed of an object may be variable over a period of time, there is a need to be able to determine its speed at a specific time instance. That is, there is a need to determine the rate of change of distance with respect to time at any time instant.

Fig. 15.5 Issac Newton

Fig. 15.6 Wilhelm Gottfried
Leibniz

The direction in which an object is moving at any instant of its flight was also
studied. For example, the direction in which a projectile is fired determines the
horizontal and vertical components of its velocity. The direction in which an object
is moving can vary from one instant to another (Fig. 15.6).

The problem of finding the maximum and minimum values of a function was also
studied e.g. the problem of determining the height that a bullet reaches when it is
fired. Other problems studied included those of determining the lengths of paths, the
areas of figures and the volume of solids.

Newton and Leibnitz showed that these problems could be solved by means of
the concept of the derivative of a function, i.e. the rate of change of one variable with
respect to another.

Rate of Change   The average rate of change and instantaneous rate of change are of
practical interest. For example, if a motorist drives 200 miles in 4 h then the average
speed is 50 miles/h, i.e. the distance travelled divided by the elapsed time. The actual
speed during the journey may vary: If the driver stops for lunch then the actual speed
is zero for the duration of lunch.

The actual speed is the instantaneous rate of change of distance with respect to time. This has practical implications as motorists are required to observe speed limits, and a speed camera may record the actual speed of a vehicle, with the driver subjected to a fine if the permitted speed limit has been exceeded. The actual speed is relevant in a car crash as speed is a major factor in road fatalities.

In calculus, the term Δx means a change in x and Δy means the corresponding change in y. The derivative of f at x is the instantaneous rate of change of f, and f is said to be differentiable at x. It is defined as:

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

In the formula, Δy is the increment $f(x + \Delta x) - f(x)$.

The average velocity of a body moving along a line in the time interval t to $t + \Delta t$ where the body moves from position $s = f(t)$ to position $s + \Delta s$ is given by:

$$V_{av} = \frac{\text{displacement}}{\text{Time travelled}} = \frac{\Delta s}{\Delta t} = \frac{f(t + \Delta t) - f(t)}{\Delta t}$$

The instantaneous velocity of a body moving along a line is the derivative of its position $s = f(t)$ with respect to t. It is given by:

$$v = \frac{ds}{dt} = \lim_{\Delta t \to 0} \frac{\Delta s}{\Delta t} = f'(t)$$

15.2.1   Rules of Differentiation

1. The derivative of a constant is 0. That is, for $y = f(x) = c$ (a constant value) we have $dy/dx = 0$.
2. $d/dx \, (f + g) = (df/dx) + (dg/dx)$
3. The derivative of $y = f(x) = x^n$ is given by $dy/dx = nx^{n-1}$.
4. If c is a constant and u is a differentiable function of x then $dy/dx = c \, du/dx$ where $y = cu(x)$.
5. The product of two differentiable functions u and v is differentiable and

$$\frac{d}{dx}(uv) = v\frac{du}{dx} + u\frac{dv}{dx}$$

6. The quotient of two differentiable functions u, v is differentiable (where $v \neq 0$) and.

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v\frac{du}{dx} - u\frac{dv}{dx}}{v^2}$$

7. *Chain Rule*: Suppose $h = g \circ f$ is the composite of two differentiable functions $y = g(x)$ and $x = f(t)$. Then h is a differentiable function of t whose derivative at each value of t is:

Fig. 15.7 Local Minima and
Maxima
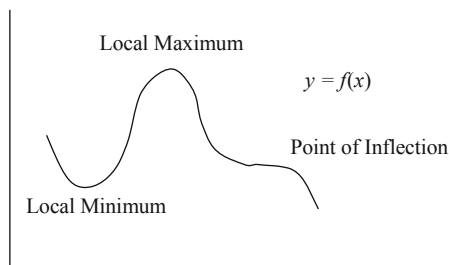
$$h'(t) = (g \circ f)'(t) = g'(f(t))f'(t)$$

This may also be written as:

$$\frac{dy}{dt} = \frac{dy}{dx}\frac{dx}{dt}$$

Derivatives of Well-Known Functions The following are the derivatives of some
well-known functions including basic trigonometric functions, the exponential
function, and the natural logarithm function.

1. $(d/dx)\sin x = \cos x$
2. $(d/dx)\cos x = -\sin x$
3. $(d/dx)\tan x = \sec^2 x$
4. $(d/dx)e^x = e^x$
5. $(d/dx)\ln x = 1/x$ (where $x > 0$)
6. $(d/dx)a^x = \ln(a)a^x$
7. $(d/dx)\log_a x = 1/x\ln(a)$
8. $(d/dx)\arcsin x = 1/\sqrt{(1-x^2)}$
9. $(d/dx)\arccos x = -1/\sqrt{(1-x^2)}$
10. $(d/dx)\arctan x = 1/(1+x^2)$

Increasing and Decreasing Functions Suppose that a function f has a derivative at
every point x of an interval I. Then

1. f increases on I if $f'(x) > 0$ for all x in I.
2. f decreases on I if $f'(x) < 0$ for all x in I.

The geometric interpretation of the first derivative test is as follows: The differentiable
functions increase on intervals where their graphs have positive slopes, and decrease
on intervals where their graphs have negative slopes (Fig. 15.7).

If f' changes from positive to negative values as x passes from left to right through
point c then the value of f at c is a *local maximum* value of f. Similarly, if f' changes
from negative to positive values as x passes from left to right through point c then
the value of f at c is a *local minimum* value of f.

The graph of a differentiable function $y = f(x)$ is concave down in an interval where
f' decreases and concave up in an interval where f' increases. This may be defined by

the second interval test for concavity. In other words, the graph of $y = f(x)$ is concave down in an interval where $f''' < 0$ and concave up in an interval where $f'' > 0$.

A point on the curve where the concavity changes is termed a point of inflection. That is, at a *point of inflection c*, we have that f''' positive on one side and negative on the other side. At the point of inflection c we have the value of the second derivative as zero, i.e. $f''(c) = 0$.

15.3   Integration

The derivative is a functional operator that takes a function as an argument and returns a function as a result. The inverse operation involves determining the original function from the known derivative. Integral calculus is the branch of calculus concerned with this problem. The integral of a function consists of all those functions that have it as a derivative.

Integration is applicable to problems involving area and volume. It is the mathematical process that allows the area of a region with curved boundaries to be determined, and it also allows the volume of a solid to be determined.

The problem of finding functions whose derivatives is known involves finding a function $y = F(x)$ whose derivative is given by the differential equation:

$$\frac{dy}{dx} = f(x)$$

The solution to this differentiable equation over the interval I is F if F is differentiable at every point of I and for every x in I we have:

$$\frac{d}{dx} F(x) = f(x)$$

Clearly, if $F(x)$ is a particular solution to $d/dx\ F(x) = f(x)$ then the general solution is given by:

$$y = \int f(x)dx = F(x) + k$$

since

$$\frac{d}{dx}(F(x) + k) = f(x) + 0 = f(x).$$

Rules of Integration

1. $\int (u'(x)dx = u(x) + k$
2. $\int (u(x) + v(x))dx = \int u(x)dx + \int v(x)dx.$
3. $\int au(x)dx = a \int u(x)dx$ (where a is a constent)
4. $\int x^n dx = \frac{x^{n+1}}{n+1} + k$ (where $n \neq -1$)

5. $\int \cos x \, dx = \sin x + k$
6. $\int \sin x \, dx = -\cos x + k$
7. $\int \sec^2 x \, dx = \tan x + k$
8. $\int e^x \, dx = e^x + k$

It is easy to check that the integration has been carried out correctly by determining the derivative of the resultant function, and verifying that it is the same as the function to be integrated.

Often, the goal will be to determine a particular solution satisfying certain conditions rather than the general solution. The general solution is first determined, and then the constant k that satisfies the particular solution is determined.

The *substitution method* of integration is often used to change an unfamiliar integral into one that is easier to evaluate. It is a useful method, and the procedure to evaluate $\int f(g(x)g'(x)dx$ where f', g' are continuous functions is as follows:

1. Substitute $u = g(x)$ and $du = g'(x)dx$ to obtain $\int f(u)du$
2. Integrate with respect to u.
3. Replace u by $g(x)$ in the result.

The method of *integration by parts* is a rule of integration that transforms the integral of a product of functions into simpler integrals. It is a consequence of the product rule for differentiation.

$$\int u \, dv = uv - \int v \, du$$

$$\int f(x)g'(x)dx = f(x)g(x) - \int f'(x)g(x)dx$$

Here, single prime indicates the first derivative.

15.3.1 Definite Integrals

The area of the region between the graph of the non-negative continuous function $y = f(x)$ on the interval $a \le x \le b$ of the x-axis is given by the definite integral (Fig. 15.8).

The sum of the area of the rectangles approximates the area under the curve and the more rectangles that are used the better the approximation.

The definition of the area of the region beneath the graph of $y = f(x)$ from a to b is defined to be the limit of the sum of the rectangle areas as the rectangles become smaller and smaller, and the number of rectangles used approaches infinity. The limit of the sum of the rectangle areas exists for any continuous function (Fig. 15.9).

The approximation of the area under the graph $y = f(x)$ between $x = a$ and $x = b$ is done by dividing the region into n strips, with each strip of uniform width given by $\Delta x = (b - a)/n$ and drawing lines perpendicular to the x-axis. Each strip is approximated with an inscribed rectangle where the base of the rectangle is on the

Fig. 15.8 Area under the curve
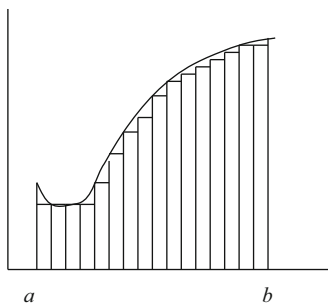
Fig. 15.9 Area under the curve—Lower Sum
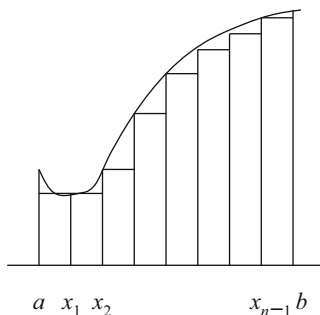
x-axis to the lowest point on the curve above. We let c_k be a point of which f takes on its minimum value in the interval from x_{k-1} to x_k and the height of the rectangle is $f(c_k)$. The sum of these areas is the approximation of the area under the curve and is given by:

$$S_n = f(c_1)\Delta x + f(c_2)\Delta x + \cdots + f(c_n)\Delta x$$

The area under the graph of a nonnegative continuous function f over the interval (a, b) is the limit of the sums of the areas of inscribed rectangles as n approaches infinity.

$$A = \lim_{n\to\infty} S_n = \lim_{n\to\infty} f(c_1)\,\Delta x + f(c_2)\,\Delta x + \cdots + f(c_n)\,\Delta x$$

$$= \lim_{n\to\infty} \sum_{k=1}^{n} f(c_k)\Delta x$$

It is not essential that the division of (a, b) into $a, x_1, x_2, \ldots, x_{n-1}, b$ gives equal subintervals $\Delta x_1 = x_1 - a$, $\Delta x_2 = x_2 - x_1$, $\ldots$ $\Delta x_n = b - x_{n-1}$. The *norm* of the subdivision is the largest interval length.

The lower Riemann sum L and the upper sum U can be formed, and the more finely divided that (a, b) is the closer the values of the lower and upper sum U and L. The upper and lower sums may be written as:

$$L = \min_1 \Delta x_1 + \min_2 \Delta x_2 + \cdots + \min_n \Delta x_n$$

$$U = \max_1 \Delta x_1 + \max_2 \Delta x_2 + \cdots + \max_n \Delta x_n$$

$$\lim_{\text{norm} \to 0} U - L = 0 \;\; (\text{i.e.} \lim_{\text{norm} \to 0} U = \lim_{\text{norm} \to 0} L)$$

Further, if $S = \Sigma f(c_k) \, \Delta x_k$ (where c_k is any point in the subinterval and $\min_k \le f(c_k) \le \max_k$ and we have:

$$L \le S \le U$$

$$\lim_{\text{norm} \to 0} U = \lim_{\text{norm} \to 0} S = \lim_{\text{norm} \to 0} L$$

Integral Existence Theorem (Riemann Integral) If f is continuous on (a, b) then

$$\int_a^b f(x)dx = \lim_{\text{norm} \to 0} \sum f(c_k) \Delta x_k$$

exists and is the same number for any choice of the numbers c_k.

Properties of Definite Integrals The following are some algebraic properties of the definite integral.

1. $\int_a^a f(x)dx = 0$
2. $\int_b^a f(x)dx = -\int_a^b f(x)dx$
3. $\int_a^b kf(x)dx = k \int_a^b f(x)dx$
4. $\int_a^b f(x)dx \ge 0 \quad$ if $f(x) \ge 0$ on (a, b)
5. $\int_a^b f(x)dx \le \int_a^b g(x)dx \quad$ if $f(x) \le g(x)$ on (a, b)
6. $\int_a^b f(x)dx + \int_b^c f(x)dx = \int_a^c f(x)dx$
7. $\int_a^b |f(x)dx + g(x)dx| = \int_a^b f(x)dx + \int_a^b g(x)dx$
8. $\int_a^b |f(x)dx - g(x)dx| = \int_a^b f(x)dx - \int_a^b g(x)dx$

15.3.2 Fundamental Theorems of Integral Calculus

We present two fundamental theorems of integral calculus.

First Fundamental Theorem: (Existence of Anti-Derivative) If f is continuous on (a, b) then $F(x)$ is differentiable at every point x in (a, b) where $F(x)$ is given by:

$$F(x) = \int_a^x f(t)dt$$

Further,

$$\frac{dF}{dx} = \frac{d}{dx} \int_a^x f(t)dt = f(x)$$

That is, if f is continuous on (a, b) then there exists *a* function $F(x)$ whose derivative on (a, b) is f.

Second Fundamental Theorem: (Integral Evaluation Theorem) If f is continuous on (a, b) and F is any anti-derivative of f on (a, b) then:

$$\int_a^b f(x)dx = F(b) - F(a)$$

That is, the procedure to calculate the definite integral of f over (a, b) involves just two steps:

1. Find an antiderivative F of f
2. Calculate $F(b) - F(a)$

For a more detailed account of integral and differential calculus the reader is referred to [Fin:88].

15.4   Numerical Analysis

Numerical analysis is concerned with devising methods for approximating solutions to mathematical problems. Often an exact formula is not available for solving a particular equation $f(x) = 0$, and numerical analysis provides techniques to approximate the solution in an efficient manner.

An algorithm is devised to provide the approximate solution. This consists of a sequence of steps to produce the solution as efficiently as possible within defined accuracy limits. The maximum error due to the application of the numerical methods needs to be determined. The algorithm is implemented in a programming language such as Fortran.

There are several numerical techniques to determine the root of an equation $f(x) = 0$. These include techniques such as the bisection method, which has been used since ancient times, and the Newton–Raphson method developed by Sir Isaac Newton (Fig. 15.10).

The bisection method is employed to find a solution to $f(x) = 0$ for the continuous function f on (a, b) where $f(a)$ and $f(b)$ have opposite signs. The method involves a repeated halving of subintervals of (a, b), with each step locating the half that contains the root. The inputs to the algorithm are the endpoints a and b, the tolerance (TOL), and the maximum number of iterations N. The steps are as follows:

1. Initialise i to 1
2. while $i \leq N$

   (i) Compute midpoint p
        $(p \rightarrow a + (b - a)/2)$
   (ii) If $f(p) = 0$ or $(b - a)/2 < $ TOL
        Output p and stop

Fig. 15.10 Bisection method
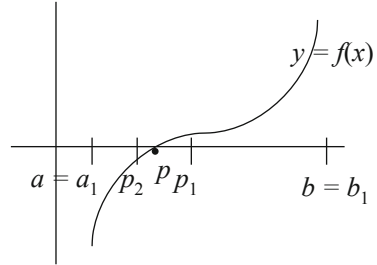
   (iii) If $f(a)f(p) > 0$
        Set endpoint $a \rightarrow p$
   (iv) Otherwise set $b \rightarrow p$
        $i \rightarrow i+1$

The Newton–Raphson method is a well-known technique to determine the roots of a function. It uses tangent lines to approximate the graph of $y = f(x)$ near the points where f is zero. The procedure for Newton's method is:

Newton's Method

1. Guess a first approximation to the root of the equation $f(x) = 0$.
2. Use the first approximation to get a second, third and so on.
3. The formula to go from the nth approximation x_n to the next approximation x_{n+1} is:

$$x_{n+1} = x_n - \frac{f(x)}{f'(x_n)}$$

where $f'(x_n)$ is the derivative of f at x_n.

    Newton's method is very efficient for calculating roots as it converges very quickly. However, the method may converge to a different root than expected if the starting value is not close enough to the root sought.

    The method involves computing the tangent line at $(x_n, f(x_n))$ and the approximation x_{n+1} is the point where the tangent intersects the x-axis.

Fixed Point Iteration and Algorithm   A fixed point of the function g is a solution to the equation $x = g(x)$. There are sufficient conditions for the existence and uniqueness of a fixed point of a function. If $g \in C(a, b)$ and $g(x) \in (a, b)$ for all $x \in (a, b)$, then g has a fixed point.

    Further, if g' exists on (a, b) and $0 < k < 1$ exists such that $|g'(x)| \leq k < 1$ for all $x \in (a, b)$ then g has a unique fixed point p in (a, b).

    The approximation of the fixed point of a function g involves choosing an initial approximation p_0 and generating the sequence $\{p_n\}_1^\infty$ by letting $p_n = g(p_{n-1})$ for each $n \geq 1$. If the sequence converges to p and g is continuous[3] then

$$p = \lim_{n \to \infty} p_n = \lim_{n \to \infty} g(p_{n-1}) = g(\lim_{n \to \infty} p_{n-1}) = g(p).$$

---

[3] For any function f that is continuous at x_0 then for any sequence $\{x_n\}$ converging on x_0 then $\lim_{x \to x_0} f(x_n) = f(x_0)$.

This technique is called fixed-point iteration and the algorithm is as follows:

1. Make an initial approximation p_0 to the fixed-point p.
2. Use the first approximation to get a second, third and so on by computing $p_i = g(p_{i-1})$.
3. Continue for a fixed number of iterations or until $|p_k - p_{k-1}| <$ TOL
4. The approximation to the fixed point is p_k if $|p_k - p_{k-1}| <$ TOL

Horner's Method and Algorithm   Horner's Method is a computationally efficient way to evaluate a polynomial function. It is named after William Horner who was a nineteenth century British mathematician and schoolmaster. Chinese mathematicians were familiar with the method in the third century. The normal evaluation of a polynomial involves computing exponentials, and this is computationally expensive. Horner's method has the advantage that fewer calculations are required, and it eliminates all exponentials by using nested multiplication and addition. It also provides an efficient way to determine the derivative of the polynomial. Consider a polynomial $P(x)$ defined by:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0$$

The Horner method to evaluate $P(x_0)$ essentially involves writing $P(x)$ as:

$$P(x) = (((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$$

The computation of $P(x_0)$ involves defining a set of coefficients b_k such that:

$$b_n = a_n$$
$$b_{n-1} = a_{n-1} + b_n x_0$$
$$\ldots\ldots$$
$$b_k = a_k + b_{k+1} x_0$$
$$\ldots\ldots$$
$$b_1 = a_1 + b_2 x_0$$
$$b_0 = a_0 + b_1 x_0$$

Then the computation of $P(x_0)$ is given by:

$$P(x_0) = b_0$$

Further, if $Q(x) = b_n x^{n-1} + b_{n-1} x^{n-2} + b_{n-2} x^{n-3} + \cdots + b_1$ then it is easy to verify that:

$$P(x) = (x - x_0)\, Q\,(x) + b_0$$

This also allows the derivative of $P(x)$ to be easily computed for x_0 since:

$$P'(x) = Q(x) + (x - x_0)\, Q'\,(x)$$

$$P'(x_0) = Q(x_0)$$

Algorithm (To evaluate polynomial and its derivative)

1. Initialise y to a_n and z to a_n (Compute b_n for P and b_{n-1} for Q)
2. For each j from $n-1$, $n-2$ to 1 compute b_j for P and b_{j-1} for Q by Set y to $x_0 y + a_j$ (i.e. b_j for P) and z to $x_0 z + y$ (i.e. b_{j-1} for Q)
3. Compute b_0 by setting y to $x_0 y + a_0$

Then $P(x_0) = y$ and $P'(x_0) = z$.

15.5   Fourier Series

Fourier series is named after Joseph Fourier, a nineteenth century French mathematician, and is used to solve practical problems in physics. A Fourier series consists of the sum of a possibly infinite set of sine and cosine functions. The Fourier series for f on the interval $0 \leq x \leq l$ defines a function f whose value at each point is the sum of the series for that value of x.

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} \left(a_m \cos \frac{m\pi x}{l} + b_m \sin \frac{m\pi x}{l} \right)$$

The sine and cosine functions are periodic functions

Note 1: (Period of Function) A function f is periodic with period $T > 0$ if $f(x+T) = f(x)$ for every value of x. The sine and cosine functions are periodic with period 2π, i.e. $\sin(x+2\pi) = \sin(x)$ and $\cos(x+2\pi) = \cos(x)$. The functions $\sin m\pi x/l$ and $\cos m\pi x/l$ have period $T = 2\,l/m$.

Note 2: (Orthogonality) Two functions f and g are said to be orhogonal on $a \leq x \leq b$ if:

$$\int_a^b f(x)g(x)dx = 0$$

A set of functions is said to be mutually orthogonal if each distinct pair in the set is orthogonal. The functions $\sin m\pi x/l$ and $\cos m\pi x/l$ where $m = 1, 2, \ldots$ form a mutually orthogonal set of functions on the interval $-l \leq x \leq l$ and they satisfy the following orthogonal relations:

$$\int_{-l}^{l} \cos \frac{m\pi x}{l} \sin \frac{n\pi x}{l} dx = 0 \quad \text{all } m, n$$

$$\int_{-l}^{l} \cos \frac{m\pi x}{l} \cos \frac{n\pi x}{l} dx = \begin{cases} 0 & m \neq n \\ l & m = n \end{cases}$$

$$\int_{-l}^{l} \sin \frac{m\pi x}{l} \sin \frac{n\pi x}{l} dx = \begin{cases} 0 & m \neq n \\ l & m = n \end{cases}$$

The orthogonality property of the set of sine and cosine functions allows the coefficients of the Fourier series to be determined. Thus, the coefficients a_n, b_n for the convergent Fourier series $f(x)$ are given by:

$$a_n = \frac{1}{l} \int_{-l}^{l} f(x) \cos \frac{n\pi x}{l} dx \quad n = 0, 1, 2, \ldots$$

$$b_n = \frac{1}{l} \int_{-l}^{l} f(x) \sin \frac{n\pi x}{l} dx \quad n = 1, 2, \ldots$$

The values of the coefficients a_n and b_n are determined from the integrals and the ease of computation depends on the particular function f involved.

$$f(x) = \frac{a_0}{2} + \sum_{m-1}^{\infty} \left(a_m \cos \frac{m\pi x}{l} + b_m \sin \frac{m\pi x}{l} \right)$$

The values of a_n and b_n depends only on the value of $f(x)$ in the interval $-l \leq x \leq l$. The terms in the Fourier series are periodic with period $2l$ and the function converges for all x whenever it converges on $-l \leq x \leq l$. Further, its sum is a periodic function with period $2l$ and therefore $f(x)$ is determined for all x by its values in the interval $-l \leq x \leq l$.

15.6 The Laplace Transform

An integral transform takes a function f and transforms it to another function F by means of an integral. Often, the objective is to transform a problem for f into a simpler problem, and then to recover the desired function from its transform F. Integral transforms are useful in solving differential equations, and an integral transform is a relation of the form:

$$F(s) = \int_{\alpha}^{\beta} K(s,t) f(t) dt$$

The function F is said to be the transform of f and the function K is called the kernel of the transformation.

The Laplace transform is named after the well-known eighteenth century French mathematician and astronomer, Pierre Laplace. The Laplace transform of f (denoted by $\mathcal{L}\{f(t)\}$ or $F(s)$) is given by:

$$\mathcal{L}\{f(x)\} = F(s) = \int_{0}^{\infty} e^{-st} f(t) dt$$

The kernel $K(s, t)$ of the transformation is e^{-st} and the Laplace transform is defined over an integral from zero to infinity. This is defined as a limit of integrals over finite intervals as follows:

$$\int_{a}^{\infty} f(t) dt = \lim_{A \to \infty} \int_{a}^{A} f(t) dt$$

Theorem (Sufficient Condition for Existence of Laplace Transform) Suppose that f is a piecewise continuous function on the interval $0 \leq x \leq A$ for any positive A and $|f(t)| \leq Ke^{at}$ when $t \geq M$ where a, K, M are constants and K, $M > 0$ then the Laplace transform $\mathcal{L}\{f(t)\} = F(s)$ exists for $s > a$.

The following examples are Laplace transforms of some well-known elementary functions.

$$\mathcal{L}\{1\} = \int_0^\infty e^{-st} dt = \frac{1}{s}, \quad s > 0$$

$$\mathcal{L}\{e^{at}\} = \int_0^\infty e^{-st} e^{at} dt = \frac{1}{s-a} \quad s > a$$

$$\mathcal{L}\{\sin at\} \int_0^\infty e^{-st} \sin at\, dt = \frac{a}{s^2 + a^2} \quad s > 0$$

15.7   Differential Equations

Many important problems in engineering and physics involve determining a solution to an equation that contains one or more derivatives of the unknown function. Such an equation is termed a differential equation, and the study of these equations began with the development of the calculus by Newton and Leibnitz.

Differential equations are classified as ordinary or partial on the basis of whether the unknown function depends on a single independent variable or on several independent variables. In the first case only ordinary derivatives appear in the differential equation and it is said to be an *ordinary differential equation*. In the second case the derivatives are partial and the equation is termed *a partial differential equation*.

For example, Newton's second law of motion ($F = ma$) expresses the relationship between the force exerted on an object of mass m and the acceleration of the object. The force vector is in the same direction as the acceleration vector. It is given by the ordinary differential equation:

$$m \frac{d^2 x(t)}{dt^2} = F(x(t))$$

The next example is that of a second-order partial differential equation. It is the wave equation and is used for the description of waves (e.g. sound, light and water waves) as they occur in physics. It is given by:

$$a^2 \frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial^2 u(x,t)}{\partial t^2}$$

There are several fundamental questions with respect to a given differential equation. First, there is the question as to the existence of a solution to the differential equation. Second, if it does have a solution, then whether this solution is unique. A third question is to how to determine a solution to a particular differential equation.

Differential equations are classified into linear or nonlinear differential equations. The ordinary differential equation $F(x, y, y', \cdots, y^{(n)}) = 0$ is said to be *linear* if F

is a linear function of the variables $y, y', \cdots, y^{(n)}$. The general ordinary differential equation is of the form:

$$a_0(x)y^{(n)} + a_1(x)y^{(n-1)} + \cdots + a_n(x)y = g(x)$$

A similar definition applies to partial differential equations and an equation is *non-linear* if it is not linear.

15.8   Review Questions

1. Explain the concept of the limit and what it means for the limit of the function f at a to be l.
2. Explain the concept of continuity.
3. Explain the difference between average velocity and instantaneous velocity, and explain the concept of the derivative of a function.
4. Determine the following
   a. $\lim\limits_{x \to x_0} \sin x$
   b. $\lim\limits_{x \to x_0} x \cos x$
   c. $\lim\limits_{x \to -\infty} |x|$
5. Determiine the derivative of the following functions
   a. $y = x^3 + 2x + 1$
   b. $y = x^2 + 1, \quad x = (t + 1)^2$
   c. $y = \cos x^2$
6. Determine the integral of the following functions
   a. $\int (x^2 - 6x)dx$
   b. $\int \sqrt{(x - 6)dx}$
   c. $\int (x^2 - 4)^2 3x^3 dx$
7. State and explain the significance of the first and second fundamental theorems of the calculus.
8. Describe Horner's method for evaluating a polynomial function and its derivative.
9. What is a periodic function? Give examples.
10. Describe the applications of Fourier series, Laplace transforms and differential equations.

15.9   Summary

This chapter provided a brief introduction to calculus including differentiation, integration, numerical analysis, Fourier series, Laplace transforms and differential equations.

Newton and Leibniz developed calculus independently in the late seventeenth century. It plays a key role in describing how rapidly things change and may be employed to calculate areas of regions under curves, volumes of figures, and in finding tangents to curves.

In calculus, the term Δx means a small change in x and Δy means the corresponding change in y. The derivative of f at x is the instantaneous rate of change of f, and is defined as:

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - (fx)}{\Delta x}$$

Integration is the inverse operation of differentiation and involves determining the original function from the known derivative. The integral of a function consists of all those functions that have the function as a derivative.

Integration is applicable to problems involving area and volume, and it allows the area of a region with curved boundaries to be determined.

Numerical analysis is concerned with devising methods for approximating solutions to mathematical problems. Often an exact formula is not available for solving a particular problem, and numerical analysis provides techniques to approximate the solution in an efficient manner.

A Fourier series consists of the sum of a possibly infinite set of sine and cosine functions. A differential equation is an equation that contains one or more derivatives of the unknown function.

This chapter has sketched some important results in calculus, and the reader is referred to [Fin:88] for more detailed information.

Chapter 16
Graph Theory

Key Topics
   Directed Graphs
   Adirected Graphs
   Incidence Matrix
   Degree of Vertex
   Walks and Paths
   Hamiltonian Path
   Graph Algorithms

16.1   Introduction

Graph theory is a practical branch of mathematics that deals with the arrangements
of certain objects known as vertices (or nodes) and the relationships between them. It
has been applied to practical problems such as the modelling of computer networks,
determining the shortest driving route between two cities, the link structure of a
website, the travelling salesman problem and the four-colour problem.[1]

Consider a map of the London underground, which is issued to users of the
underground transport system in London. Then this map does not represent every
feature of the city of London, as it includes only material that is relevant to the users
of the London underground system. In this map, the exact geographical location
of the stations is unimportant, and the essential information is how the stations are
interconnected to one another, as this allows a passenger to plan a route from one
station to another. That is, the map of the London underground is essentially a model
of the transport system that shows how the stations are interconnected.

---

[1] The four-colour theorem states that given any map it is possible to colour the regions of the map
with not more than four colours such that no two adjacent regions have the same colour.

Fig. 16.1 Königsberg seven
bridges problem
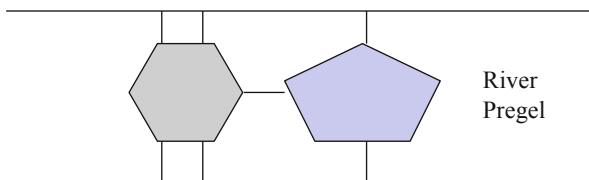
River
Pregel

Fig. 16.2 Königsberg graph
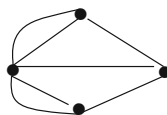
The seven bridges of Königsberg[2] is one of the earliest problems in graph theory. The city was set on both sides of the Pregel River in the early eighteenth century, and it consisted of two large islands that were connected to each other and the mainland by seven bridges. The problem was to find a walk through the city that would cross each bridge once and once only (Fig. 16.1).

Euler showed that the problem had no solution, and his analysis helped to lay the foundations for graph theory as a discipline. He noted, in effect, that for a walk through a graph traversing each edge exactly once depends on the *degree* of the nodes (i.e. the number of edges touching it). He showed that a necessary and sufficient condition for the walk is that the graph is connected and has zero or two nodes of odd degree. For the Königsberg graph, the four nodes (i.e. the land masses) have odd degree (Fig. 16.2).

A *graph* is a collection of objects that are interconnected in some way. The objects are typically represented by vertices (or nodes), and the interconnections between them are represented by edges (or lines). We distinguish between directed and adirected graphs, where a *directed graph* is mathematically equivalent to a binary relation, and an *adirected graph* is equivalent to a symmetric binary relation.

16.2 Undirected Graphs

An *undirected graph* (*adirected graph*) G is a pair of finite sets (V, E) such that E is a binary symmetric relation on V. The set of vertices (or nodes) is denoted by V(G) and the set of edges is denoted by (E(G); Fig. 16.3).

A *directed graph* is a pair of finite sets (V, E) where E is a binary relation (that may not be symmetric). A *directed acyclic graph* (*dag*) is a directed graph that has no

---

[2] Königsberg was founded in the thirteenth century by Teutonic knights and was one of the cities of the Hanseatic League. It was the historical capital of East Prussia (part of Germany), and it was annexed by Russia at the end of the Second World War. The German population either fled the advancing Red army or were expelled by the Russians in 1949. The city is now called Kaliningrad. The famous German philosopher, Immanuel Kant, spent all his life in the city, and is buried there.

Fig. 16.3 Undirected graph
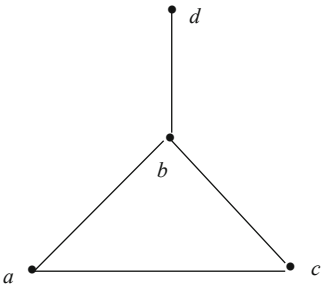
Fig. 16.4 Directed graph
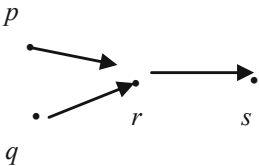
cycles. The example above is of a directed graph with three edges and four vertices (Fig. 16.4).

An edge $e \in E$ consists of a pair $<x, y>$ where x, y are adjacent nodes in the graph. The *degree* of x is the number of nodes that are adjacent to x. The set of edges is denoted by $E(G)$, and the set of vertices is denoted by $V(G)$.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *weighted graph* is a graph $G = (V, E)$ together with a weighting function $w : E \rightarrow N$ which associates a weight with every edge in the graph. A weighting function may be employed in modelling computer networks: for example, the weight of an edge may be applied to model the bandwidth of a telecommunications link between two nodes.

For an adirected graph, the weight of the edge is the same in both directions: i.e. $w(v_i, v_j) = w(v_j, v_i)$ for all edges $< v_i, v_j >$ in the graph G.

Two vertices x, y are adjacent if $xy \in E$, and x and y are said to be incident to the edge xy. A matrix may be employed to represent the adjacency relationship.

Example Consider the graph $G = (V, E)$ where $E = \{u = ab, v = cd, w = fg, x = bg, y = af\}$.
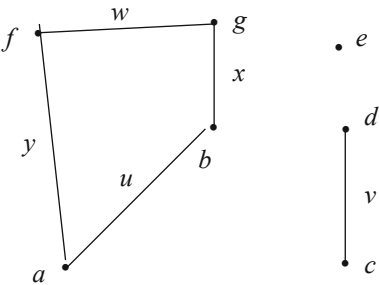
Fig. 16.5 Adjacency matrix

	a	b	c	d	e	f	g
a	0	1	0	0	0	1	0
b	1	0	0	0	0	0	1
c	0	0	0	1	0	0	0
d	0	0	1	0	0	0	0
e	0	0	0	0	0	0	0
f	1	0	0	0	0	0	1
g	0	1	0	0	0	1	0

Fig. 16.6 Incidence matrix

	u	v	w	x	y
a	1	0	0	0	1
b	1	0	0	1	0
c	0	1	0	0	0
d	0	1	0	0	0
e	0	0	0	0	0
f	0	0	1	0	1
g	0	0	1	1	0

An adjacency matrix may be employed to represent the relationship of adjacency in a graph. Its construction involves listing the vertices in the rows and columns, and an entry of 1 is made in the table if the two vertices are adjacent and 0 otherwise (Fig. 16.5).

Similarly, we can construct a table describing the incidence of edges and vertices by constructing an incidence matrix. The incidence matrix for the example above is (Fig. 16.6):

Two graphs $G = (V, E)$ and $G' = (V', E')$ are said to be isomorphic if there exists a bijection $f: V \rightarrow V'$ such that for any $u, v \in V$, $uv \in E$ if and only if $f(u)f(v) \in E'$. The mapping f is called an isomorphism. Two graphs that are isomorphic are essentially equivalent apart from a re-labelling of the nodes and edges.

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs, then G' is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. Given $G = (V, E)$ and $V' \subseteq V$, then we can induce a subgraph $G' = (V', E')$ by restricting G to V'. (denoted by $G_{|V'|}$). The set of edges in E' is defined as:

$$E' = \{e \in E : e = uv \text{ and } u, v \in V'\}$$

The *degree* of a vertex v is the number of distinct edges incident to v. It is denoted by deg v where:

$$\text{deg } v = |\{e \in E : e = vx \text{ for some } x \in V\}|$$

$$= |\{u \in V : vu \in E\}|$$

A vertex of degree 0 is called an isolated vertex.

Theorem 16.1 *Let $G = (V, E)$ be a graph then*

$$\sum\nolimits_{v \in V} \deg\ v = 2\,|E|.$$

Proof This result is clear, since each edge contributes one to each of the vertex degrees. The formal proof is by induction based on the number of edges in the graph, and the basis case is for a graph with no edges (i.e. where every vertex is isolated), and the result is immediate.

The inductive step is to assume that the result is true for all graphs with k or fewer edges. We then consider a graph $G = (V, E)$ with $k + 1$ edges.

Choose an edge $e = xy \in E$ and consider the graph $G' = (V, E')$, where $E' = E \backslash \{e\}$. Then, G' is a graph with k edges and therefore letting $\deg'\ v$ represent the degree of a vertex in G' we have:

$$\sum\nolimits_{v \in V} \deg'\ v = 2\,\big|E'\big| = 2(\,|E| - 1).$$

The degree of x and y are one less in G' than they are in G. That is,

$$\sum\nolimits_{v \in V} \deg v - 2 = 2(|E| - 1)$$

$$\Rightarrow \sum\nolimits_{v \in V} \deg v = 2(|E|)$$

A graph $G = (V, E)$ is said to be *complete* if all the vertices are adjacent: i.e. $E = V \times V$.

A common problem encountered in graph theory is determining whether or not there is a route from one vertex to another. Often, once a route has been identified the problem then becomes that of finding the shortest or most efficient route to the destination vertex.

Consider a person walking in a forest from A to B where the person does not know the way to B. Often, the route taken will involve the person wandering around aimlessly and often retracing parts of the route until eventually the destination B is reached. This is an example of a *walk* from v_1 to v_k where there is repetition of edges.

If all of the edges of a walk are distinct, then it is called a *trail*. A *path* v_1, v_2, $\ldots$, v_k from vertex v_1 to v_k is of length $k - 1$ and consists of the sequence of edges $<v_1, v_2>$, $<v_2, v_3>$, $\ldots <v_k - 1, v_k>$ where each $<v_i, v_{i+1}>$ is an edge in E. The vertices in the path are all distinct apart from possibly v_1 and v_k. The path is said to be a cycle if $v_1 = v_k$. A graph is said to be *acyclic* if it contains no cycles.

Theorem 16.2 *Let $G = (V, E)$ be a graph and $W = v_1, v_2 \ldots, v_k$ be a walk from v_1 to v_k. Then there is a path from v_1 to v_k using only the edges of W.*

Proof The walk W may be reduced to a path by successively replacing redundant parts in the walk of the form $v_i\ v_{i+1} \ldots, v_j$ where $v_i = v_j$ with $v_i\ v_j$. That is, we successively remove cycles from the walk and this clearly leads to a path (not necessarily the shortest path) from v_1 to v_k.

Theorem 16.3 *Let $G = (V, E)$ be a graph and let $u, v \in V$ with $u \neq v$. Suppose that there exists two different paths from u to v in G, then G contains a cycle.*
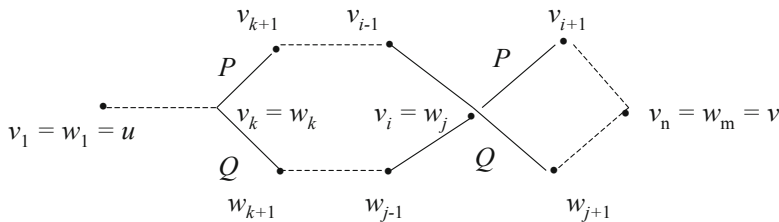
Suppose that $P = v_1, v_2, \ldots, v_n$ and $Q = w_1, w_2 \ldots, w_m$ are two distinct paths from u to v (where $u \neq v$), and $u = v_1 = w_1$ and $v = v_n = w_m$. Suppose P and Q are identical for the first k vertices (k could be 1), and then differ (i.e. $v_{k+1} \neq w_{k+1}$). Then, Q crosses P again at $v_n = w_m$ and possibly several times before then. Suppose the first occurrence is at $v_i = w_j$ with $k < i \leq n$. Then, $w_k, w_{k+1}, w_{k+2}, \ldots, w_j \; v_{i-1}, v_{i-2}, \ldots, v_k$ is a closed path (i.e. a cycle), since the vertices are all distinct.

If there is a path from v_1 to v_2, then it is possible to define the *distance* between v_1 and v_2. This is defined to be the total length (number of edges) of the shortest path between v_1 and v_2.

16.2.1   Hamiltonian Paths

A *Hamiltonian path*[3] in a graph $G = (V, E)$ is a path that visits every vertex once and once only. In another words, the length of a Hamiltonian path is $|V| - 1$. A graph is Hamiltonian connected if for every pair of vertices there is a Hamilitonian path between the two vertices.

Hamilitonian paths are applicable to the travelling salesman problem, where a salesman wishes to travel to k cities in the country without visiting any city more than once. There are several sufficient conditions for the existence of a Hamiltonian path.

Theorem 16.4 *Let $G = (V, E)$ be a graph with $|V| = n$ and such that deg $v +$ deg $w \geq n - 1$ for all non-adjacent vertices v and w. Then, G possesses a Hamilitonian path.*

Proof The proof involves first showing that G is connected and then considering the largest path in G of length $k - 1$ and assuming that $k < n$. A contradiction is then derived and it is deduced that $k = n$. A proof is in [Pif:91].

---

[3] These are named after Sir William Rowan Hamilton who was a nineteenth century Irish mathematician and astronomer.

16.3 Trees

A graph is said to be *connected* if for any two given vertices v_1, v_2 in V there is a path from v_1 to v_2. An acylic graph is termed a *forest* and a connected forest is termed a *tree*.

A graph G is a tree if and only if for each pair of vertices in G there exists a unique path in G joining these vertices. This is since G is connected and acyclic, with the connected property giving the existence of at least one path and the acylic property giving uniqueness.

Theorem 16.5 *Let $G = (V, E)$ be a tree and let $e \in E$ then $G' = (V, E \backslash \{e\})$ is disconnected and has two components.*

Proof Let $e = uv$, then since G is connected and acyclic uv is the unique path from u to v, and thus G' is disconnected since there is no path from u to v in G'.

It is thus clear that there are at least two components in G' with u and v in different components. We show that any other vertex w is connected to u or to v in G'.

Since G is connected, there is a path from w to u in G, and if this path does not use e then it is in G' as well, and therefore u and w are in the same component of G'.

If it does use e, then e is the last edge of the graph since u cannot appear twice in the path, and so the path is of the form $w, \ldots, v, u$ in G. Therefore, there is a path from w to v in G', and so w and v are in the same component in G'. Therefore, there are only two components in G'.

Theorem 16.6 *Let $G = (V, E)$ be a connected graph, then G is a tree if and only if $|E| = |V| - 1$.*

Proof This result may be proved by induction on the number of vertices $|V|$ and the application of theorem 16.5.

16.3.1 Binary Trees

A *binary tree* is a tree in which each node has at most two child nodes (termed left and right child nodes). A node with children is termed a *parent node*, and the top node of the tree is termed the root node. Any node in the tree can be reached by starting from the root node, and by repeatedly taking either the left branch (left child) or right branch (right child) until the node is reached. Binary trees are used in computing to implement efficient searching algorithms (Fig. 16.7).

The *depth* of a node is the length of the path (i.e. the number of edges) from the root to the node. The depth of a tree is the length of the path from the root to the deepest node in the tree. A *balanced* binary tree is one in which the depth of the two subtrees of any node never differs by more than 1.

The root of the binary tree in Fig. 16.7 is A and its depth is 3. The tree is unbalanced and unsorted.

Tree traversal is a systematic way of visiting each node in the tree exactly once, and we distinguish between *breadth first search* in which every node on a particular level is visited before going to a lower level, and *depth first search* where one starts

Fig. 16.7 Binary tree
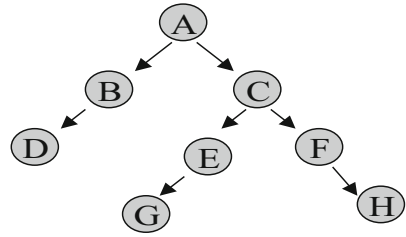
at the root and explores as far as possible along each branch before backtracking.
The traversal in depth first search may be in pre-order, in-order or post-order.

16.4   Graph Algorithms

Graph algorithms are employed to solve various problems in graph theory includ-
ing network cost minimisation problems; shortest path algorithms; longest path
algorithms and timetable construction problems.

A length function $l: E \rightarrow \mathbb{R}$ may be defined on the edges of a connected graph
$G = (V, E)$, and a shortest path from u to v in G is a path P with edge set E' such that
$l(E')$ is minimal.

Dijkstra's shortest path algorithm is described in [Pif:91].

16.5   Review Questions

1. What is a graph and explain the difference between an adirected graph and
   a directed graph.
2. Determine the adjacency and incidence matrices of the following graph
   where $V = \{a, b, c, d, e\}$ and $E = \{ab, bc, ae, cd, bd\}$.
3. Determine if the two graphs G and G' defined below are isomorphic:
   a. $G = (V, E)$, $V = \{a, b, c, d, e, f, g\}$ and $E = \{ab, ad, ae, bd, ce, cf, dg,$
      $fg, bf\}$.
   b. $G' = (V', E')$, $V' = \{a, b, c, d, e, f, g\}$ and $E' = \{ab, bc, cd, de, ef, fg, ga,$
      $ac, be\}$.

16.6   Summary

This chapter provided a brief introduction to graph theory, which is a practical branch
of mathematics that deals with the arrangements of vertices and edges between them.
It has been applied to practical problems such as the modelling of computer networks,

determining the shortest driving route between two cities and the travelling salesman problem.

An undirected graph G is a pair of finite sets (V, E) such that E is a binary symmetric relation on V, whereas a directed graph is a binary relation that is not symmetric. An adjacency matrix is used to represent whether two vertices are adjacent to each other, whereas an incidence matrix indicates whether a vertex is part of a particular edge.

A tree is a connected and acylic graph, and a binary tree is a tree in which each node has at most two child nodes.

References

[Ack:94]   Ackrill, J.L.: Aristotle the Philosopher. Clarendon, Oxford (1994)
[Ada:84]   Adams, E.: Optimizing preventive service of software products. IBM Res J. **28**(1), 2–14, (1984).
[AnL:94]   Anglin, W.S., Lambek, J.: The Heritage of Thales. Springer Verlag, New York (1995)
[Bab:11]   Baber, R.L.: The Language of Mathematics. Utilizing Math in Practice, Wiley, New York (2011)
[Bec:00]   Beck, K.: Extreme Programming Explained. Embrace Change. Addison Wesley (2000).
[BjJ:78]   Bjørner, D., Jones, C.: The Vienna Development Method. The Meta language. Lecture Notes in Computer Science 61. Springer Verlag, New York (1978)
[BjJ:82]   Bjørner, D., Jones, C.: Formal Specification and Software Development. Prentice Hall International Series in Computer Science. Prentice Hall (1982)
[BoM:79]   Boyer, R., Moore, J.S.: A Computational Logic. The Boyer Moore Theorem Prover. Academic, New York (1979)
[Boe:88]   Boehm, B.: A spiral model for software development and enhancement. Computer. **21**(5), 61–75, May (1988)
[Brk:75]   Brooks, F.: The Mythical Man Month. Addison Wesley, Menlo Park (1975)
[Brk:86]   Brooks, F.: No Silver Bullet. Essence and Accidents of Software Engineering. In: Information Processing. Elsevier, Amsterdam (1986)
[Bro:90]   Brown, M.J.D.: Rationale for the development of the UK Defence Standards for Safety Critical software. Compass Conference, London, 25–28 June 1990
[BuF:89]   Richard, L.B., Faires, J.D.: Numerical Analysis, 4th edn. PWS Kent, Boston (1989)
[CKS:11]   Chrissis, M.B., Conrad, M., Shrum, S.: CMMI. Guidelines for Process Integration and Product Improvement, 3rd edn. SEI Series in Software Engineering. Addison Wesley, London (2011)
[CoM:90]   Cobb, R.H., Mills, H.D.: Engineering software under statistical quality control. IEEE Softw. **7**(6), 44–54 (1990)
[Crs:79]   Crosby, P.: Quality is Free. The Art of Making Quality Certain. McGraw Hill, New York (1979)
[Dem:86]   Deming, W.E.: Out of Crisis. M.I.T. Press, Cambridge (1986)
[Dij:76]   Dijkstra, E.W.: A Disciple of Programming. Prentice Hall (1976)
[Dil:90]   Diller, A.: Z. An Introduction to Formal Methods. Wiley, England (1990)
[Fag:76]   Fagan, M.: Design and code inspections to reduce errors in software development. IBM Syst. J. **15**(3), 182–211 (1976)
[Fen:95]   Fenton, N.: Software Metrics: A Rigorous Approach. Thomson Computer Press, Boston (1995)
[Fin:88]   Finney, T.: Thomas, G.B.: Calculus and Analytic Geometry. 7th edn. Addison Wesley, Boston (1988)
[Flo:63]   Floyd, R.: Syntactic analysis and operator precedence. JACM. **10**, 316–333 (1963)

[Flo:64]   Floyd, R.: The syntax of programming languages. A survey. IEEE Trans. Electronic
             Comput. EC-**13**(4), 346–353 (1964)

[Flo:67]   Floyd, R.: Assigning meanings to programs. Proc. Symp. Appl. Math. **19**, 19–32
             (1967)

[Ger:94]   Gerhart, S., Craighen, D., Ralston, T.: Experience with Formal Methods in Critical
             Systems. IEEE Software, January (1994).

[Glb:76]   Gilb, T.: Software Metrics. Winthrop Publishers, Cambridge (1976)

[Glb:94]   Gilb, T., Graham, D.: Software Inspections. Addison Wesley, Menlo Park (1994)

[Gri:81]   Gries, D.: The Science of Programming. Springer Verlag, Berlin (1981)

[HB:95]   Hinchey, M., Bowen, J. (eds.): Applications of Formal Methods. Prentice Hall
             International Series in Computer Science, Prentice Hall (1995)

[Hea:56]   Heath, S.T.: Euclid. The Thirteen Books of the Elements, vol. 1. Dover Publications,
             New York (1956) (First published in 1925)

[Hey:66]   Heyting, A.: Intuitionist Logic. An Introduction. North-Holland Publishing, Amster-
             dam (1966)

[Hor:69]   Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM.
             12(10), 576–585 (1969)

[Hor:85]   Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International
             Series in Computer Science. Prentice Hall (1985)

[HoU:79]   Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and
             Computation. Addison-Wesley, Boston (1979)

[Hum:89]   Humphry, W.: Managing the Software Process. Addison Wesley, Boston (1989).

[Jon:86]   Jones, C.: Systematic Software Development using VDM. Prentice Hall International.
             Prentice Hall (1986)

[Jur:00]   Juran, J.: Juran's Quality Handbook, 5th edn. McGraw Hill, New York (2000)

[Kel:97]   Kelly, J.: The Essence of Logic. Prentice Hall. (1997)

[Knu:97]   Knuth, D.: The Art of Computer Programming, vol. 1, 3rd edn. Addison Wesley,
             Boston (1997)

[Kuh:70]   Kuhn, T.: The Structure of Scientific Revolutions. University of Chicago Press,
             Chicago (1970)

[Lak:76]   Lakatos, I.: Proof and Refutations. The Logic of Mathematical Discovery. Cambridge
             University Press, Cambridge (1976)

[Lof:84]   Löf, P.M.: Intuitionist Type Theory. Notes by Giovanni Savin of lectures given in
             Padua, June, 1980. Bibliopolis, Napoli (1984)

[Mac:90]   Mac, M.A.A.: Computation models and computing. PhD Thesis. Department of
             Computer Science. Trinity College Dublin, Dublin (1990)

[McD:94]   McDonnell, E.: MSc. Thesis. Department of Computer Science. Trinity College
             Dublin, Dublin. (1994)

[Men:87]   Mendelson, E.: Introduction to Mathematical Logic. Wadsworth and Cole/Brook,
             Advanced Books & Software, Monterey (1987)

[Mey:90]   Meyer, B.: Introduction to the Theory of Programming Languages. Prentice Hall
             (1990)

[Mil:89]   Milner, R., et al.: A Calculus of Mobile Processes. Part 1. LFCS Report Series.
             ECS-LFCS-89-85. Department of Computer Science. University of Edinburgh, UK
             (1989)

[MOD:91a]   Ministry of Defence: 00-55 (Part 1)/Issue 1. The Procurement of Safety Critical Soft-
             ware in Defence Equipment. Part 1: Requirements. Interim Defence Standard, UK
             (1991a)

[MOD:91b]   Ministry of Defence: 00-55 (Part 2)/Issue 1. The Procurement of Safety Critical Soft-
             ware in Defence Equipment. Part 2: Guidance. Ministry of Defence. Interim Defence
             Standard, UK (1991b)

[Nau:60]   Naur, P.: Report on the algorithmic language, ALGOL 60. Commun. ACM. **3**(5),
             299–314 (1960)

[Nbs:77]   National Bureau of Standards: Data Encryption Standard. FIPS-Pub 46. U.S. Department of Commerce, Washington, DC, Jan (1977)

[NFK:07]   Neubauer, A., Freunderberger, J., Kühn, V.: Coding Theory. Algorithms, Architectures and Applications. Wiley, Chichester (2007)

[ORg:97]   O'Regan, G.: Modelling Organizations and Structures in the Real World. PhD Thesis. Trinity College Dublin, Dublin (1997)

[ORg:02]   O'Regan, G.: A Practical Approach to Software Quality. Springer Verlag, New York (2002)

[ORg:06]   O'Regan, G.: Mathematical Approaches to Software Quality. Springer, London (2006)

[ORg:10]   O'Regan, G.: Introduction to Software Process Improvement. Springer, London (2010)

[ORg:12]   O'Regan, G.: A Brief History of Computing, 2nd edn., Springer, London (2012)

[Par:01]   Parnas, D.L.: Software Fundamentals. In: Hoffman, D., Weiss, D. (eds.) Addison Wesley, Longman (2001)

[Par:72]   Parnas, D.: On the criteria to be used in decomposing systems into modules. Commun. ACM. **15**(12), 1053–1058 (1972)

[Par:93]   Parnas, D.L.: Predicate calculus for software engineering. IEEE Trans. Softw. Eng. **19**(9), 856–862 (1993)

[Pif:91]   Piff, M.: Discrete Mathematics. An Introduction for Software Engineers. Cambridge University Press, Cambridge (1991)

[Plo:81]   Plotkin, G.: A Structural Approach to Operational Semantics. Technical Report DAIM FN-19. Computer Science Department. Aarhus University, Denmark (1981)

[PoH:76]   Pohlig, S., Hellman, M.: An Improved algorithm for computing algorithms over GF(p) and its cryptographic significance. IEEE Trans. Inf. Theory. **24**, 106–110 (1978)

[Pol:57]   Polya, G.: How to Solve It. A New Aspect of Mathematical Method. Princeton University Press, Princeton (1957)

[Pri:59]   de Solla Price, D.J.: An ancient greek computer. Sci. Am. **200**(6), 60–67. June (1959)

[Res:84]   Resnikoff, H.L., Wells, R.O.: Mathematics in Civilisation. Dover Publications, New York (1984)

[RoS:94]   Rozenberg, G., Salomaa, A.: Cornerstones of Undecideability. Prentice Hall. (1994)

[Roy:70]   Royce, W.: The Software Lifecycle Model (Waterfall Model). In Proc. WESTCON, August (1970)

[Rum:99]   Rumbaugh, J., et al.: The Unified Modelling Language. User Guide. Addison Wesley, Boston (1999)

[Sha:48]   Shannon, C.: A Mathematical Theory of Communication. Bell Systems Technical J. **27**, 379–423 (1948)

[Smi:23]   Smith, D.E.: The History of Mathematics. Dover Publications, New York (1923)

[Spi:92]   Spivey, J.M.: The Z Notation. A Reference Manual. Prentice Hall International Series in Computer Science (1992)

[Std:99]   Standish Group Research Note: Estimating: Art or Science. Featuring Moritz Cost Expert (1999)

[Sto:77]   Stoy, J.: Denotational Semantics. The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge (1977)

[Tie:91]   Tierney, M.: The Evolution of Def Stan 00-55 and 00-56. An intensification of the formal methods debate in the UK. Research Centre for Social Sciences. University of Edinburgh, Edinburgh (1991)

[Yan:98]   Yan, S.Y.: Number Theory for Computing, 2nd edn. Springer, Berlin (1998)

[Wic:00]   Wichmann, B.A.: A Personal View of Formal Methods. National Physical Laboratory, Teddington, March (2000)

Glossary

AECL	Atomic Energy Canada Ltd.
AES	Advanced Encryption Standard
AMN	Abstract Machine Notation
BCH	Bose, Chauduri and Hocquenghem
BNF	Backus Naur Form
CCS	Calculus Communicating Systems
CICS	Customer Information Control System
CMM	Capability Maturity Model
CMMI®	Capability Maturity Model Integration
CSP	Communicating Sequential Processes
DES	Data Encryption Standard
DOD	Department of Defence
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
FSM	Finite State Machine
GCHQ	General Communications Headquarters
GSM	Global System Mobile
HOL	Higher Order Logic
IBM	International Business Machines
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standards Organization
LPF	Logic of Partial Functions
MTBF	Mean time between failure
MTTF	Mean time to failure
MOD	Ministry of Defence
NATO	North Atlantic Treaty Organization
NIST	National Institute of Standards &Technology
NBS	National Bureau of Standards
OMT	Object Modelling Technique
PMP	Project Management Professional
QA	Quality Assurance

RSA	Rivest, Shamir and Adleman
RUP	Rational Unified Process
SCAMPI	Standard CMM Appraisal Method for Process Improvement
SECD	Stack, Environment, Code, Dump
SEI	Software Engineering Institute
SPICE	Software Process Improvement and Capability Determination
SQA	Software Quality Assurance
UML	Unified Modelling Language
UMTS	Universal Mobile Telecommunications System
VDM	Vienna Development Method
VDM♣	Irish School of VDM
XP	Extreme Programming

Index